ORIC-ATMOS ORIC-1

PROGRAMMER EN LANGAGE MACHINE SUR ORIC-ATMOS ET ORIC-1

Bruce Smith



cedic/nathan

Programmer en langage machine sur Oric-Atmos et Oric-1

Bruce Smith

Adapté de l'anglais

par Yvette Fischer

Editions Cedic
6, 10, boulevard Jourdan, 75014 - Paris
Tél. : (1) 565.06.06

L'édition originale *Machine Code for the Atmos and Oric-1*
a été publié par Shiva Publishing Limited.

Ce volume porte la référence
ISBN 2-7124-0574-9

Sommaire

Préface

Au coeur de votre micro-ordinateur Oric-Atmos ou Oric-1 se trouve une puce minuscule qu'on appelle le microprocesseur 65Ø2. C'est elle qui se charge de coordonner et de contrôler tout ce que fait l'Oric tant qu'il est sous tension, et c'est une lourde tâche! Contrairement à ce que vous pourriez penser, programmer ce microprocesseur dans son propre langage — le langage machine — n'est pas difficile et le but de ce livre est justement de vous appprendre à le faire.

Ce livre suppose que vous avez quelques notions du BASIC de l'Oric, et que vous ne savez absolument rien sur le langage machine, bien que vous ayez certainement déjà lu le chapitre consacré à ce sujet dans le *Manuel de l'Oric Atmos* ou le *Manuel de programmation du BASIC de l'Oric-1*. Je me suis efforcé d'ordonner logiquement lss différents chapitres et d'éviter un vocabulaire trop technique.

Plutôt que de consacrer chaque chapitre à une question spécifique, j'ai préféré introduire les nouveaux concepts petit à petit, lorsque le besoin s'en faisait sentir. Dans la mesure du possible, j'ai donné des exemples de programmes pour faciliter l'assimilation du point traité. Dans la plupart des cas, ces exemples sont commentés et chaque instruction est analysée et expliquée.

Programmer en code machine sur l'Atmos et l'Oric-1 se suffit à lui-même : il comprend une description complète de toutes les instructions disponibles en langage machine et suggère quelques applications. Après les premiers chapitres, qui sont un peu théoriques, j'ai introduit les registres du 65Ø2 et j'ai expliqué comment, où et quand entrer les routines en langage machine. On trouvera aussi un petit programme facilitant la saisie de telles routines.

Après avoir montré comment on peut connaître l'état du 65Ø2, j'ai présenté quelques modes d'adressage, puis l'addition et la soustraction en langage machine et les méthodes les plus simples pour manipuler et sauvegarder des données que le programme et le processeur pourront utiliser par la suite. Le chapitre sur les boucles en langage machine (comparables aux FOR...NEXT...STEP du BASIC) montre comment des séquences de codes peuvent être répétées.

Puis j'ai exposé comment utiliser les sous-programmes et les sauts (comme en BASIC avec les instructions GOSUB et GOTO). On trouvera aussi un aperçu de procédures plus complexes telles que la multiplication et la division, faisant appel à des instructions de décalage et de rotation.

Enfin, les annexes constituent des documents de référence complets et pratiques où vous pourrez trouver rapidement tout ce dont vous aurez besoin lorsque vous écrirez vos propres programmes en langage machine!

Highbury, novembre 1983

Bruce Smith

Chapitre 1

Code machine ou langage d'assemblage ?

Le microprocesseur 65Ø2 de votre Oric peut exécuter 152 opérations différentes, chacune étant définie par un nombre entier compris entre Ø et 255, appelé code opérateur (voir annexe 5). Pour créer un programme en langage machine, il nous suffit d'écrire à des emplacements successifs en mémoire les codes opérateurs adéquats. Par exemple, pour ranger la valeur 5 à l'adresse #15ØØ (autrement dit pour faire l'équivalent du BASIC POKE #15ØØ, 5), nous devons écrire les octets suivants en mémoire :

```
#A9
#Ø5
#8D
#ØØ
#15
```

et ensuite demander au 65Ø2 de l'Oric de les exécuter. Tout ceci n'est pas très clair ! C'est là que le langage d'assemblage intervient. Il nous permet d'écrire du code machine sous une forme abrégée conçue pour représenter ce que chaque code opérateur va réellement faire. Ces formes abrégées s'appellent *des mnémoniques* ; elles sont les éléments de base des programmes écrits en langage d'assemblage. Nous pouvons réécrire les codes précédents en langage d'assemblage :

```
LDA @5
STA #15ØØ
```

Ceci se lit :
charger (LoaD en anglais) l'Accumulateur avec la valeur 5 ;
ranger (STore en anglais) le contenu de l'Accumulateur à l'adresse #15ØØ.

Comme vous pouvez le voir d'après les majuscules, le mnémonique utilise certaines lettres qui rappellent le sens de l'instruction et facilitent la compréhension. Quand le programme en langage d'assemblage est terminé, il peut être traduit en code machine de deux façons.

1. Au moyen d'un assembleur. C'est un programme (écrit en code machine ou en BASIC) qui transforme l'ensemble des mnémoniques (appelé «programme source») en code machine (appelé «code objet») et les écrit en mémoire. Il ne fait aucun doute qu'un assembleur pour l'Oric, dont le succès ne fait que grandir, ne tardera pas à être disponible sur le marché.

2. En cherchant dans une table les valeurs des codes et en les écrivant en mémoire à l'aide d'un chargeur ou à l'aide d'une boucle de lecture de DATA. Cette méthode est exposée en détail dans le chapitre 5 qui contient aussi un exemple simple de chargeur.

Tous les programmes de ce livre sont présentés sous forme de code machine *et* de mnémoniques, ainsi ils peuvent être entrés avec l'une ou l'autre des méthodes précédentes. L'annexe 3 décrit de façon exhaustive les codes opérateurs du 65Ø2; ne vous inquiétez donc pas si tout ceci vous semble un peu obscur pour le moment : nous allons bientôt remédier à cela!

Pourquoi utiliser le code machine?

Une question revient souvent. «Pourquoi s'ennuyer à programmer en code machine?» En fait, il y a deux avantages à cela.

Premièrement la vitesse. Le code machine s'exécute beaucoup plus rapidement qu'un langage interprété de haut niveau tel que le BASIC. Souvenez-vous que l'interpréteur BASIC est lui-même écrit en code machine et que les commandes et les instructions du BASIC sont simplement des pointeurs vers les routines en code machine de la ROM qui exécutent les opérations demandées. C'est parce que chaque instruction et chaque commande doit d'abord être reconnue puis recherchée en ROM que la vitesse d'exécution diminue.

Deuxièmement, apprendre le code machine vous permet de comprendre comment votre ordinateur fonctionne et vous permet de créer des routines spécialisées qu'il serait impossible d'écrire en BASIC, à cause des contraintes imposées par son jeu d'instructions limité. Avec le langage machine vous dominez votre Oric plus qu'il ne vous domine!

Chapitre 2

Les nombres

Les représentations binaire, hexadécimale et décimale

Nous avons vu que les instructions exécutées par l'Oric sont représentées par des suites de nombres. Mais sous quelle forme ces nombres sont-ils rangés dans la mémoire? Sans vouloir vous déconcerter avec les merveilles de l'informatique, disons pour simplifier que les instructions sont codées en mémoire sous forme de nombres binaires. Les nombres décimaux sont des combinaisons des dix chiffres : ∅, 1, 2, 3, 4, 5, 6, 7, 8, 9. Dans ce cas, on dit qu'on utilise la *base* 1∅.

Comme leur nom l'indique, les nombres binaires utilisent la base 2, c'est-à-dire que ce sont des combinaisons de ∅ et de 1. Ces deux chiffres symbolisent les deux états électriques possibles à l'intérieur de l'Oric, à savoir ∅ volt (off) et 5 volts (on). Les codes machine dont il a été question dans le chapitre 1 sont donc représentés en mémoire de la façon suivante :

Mnémoniques	Code machine	Binaire
LDA	A9	1∅1∅1∅∅1
@#5	∅5	∅∅∅∅∅1∅1
STA	8D	1∅∅∅11∅1
∅∅	∅∅	∅∅∅∅∅∅∅∅
#15	15	∅∅∅1∅1∅1

Comme on peut le voir, chaque code machine est représenté par un ensemble de huit chiffres binaires, ou huit *bits*. Cet ensemble s'appelle un *octet*. On convient de numéroter les bits d'un octet de la façon suivante :

7	6	5	4	3	2	1	∅

Ces numéros sont disposés en ordre croissant de droite à gauche. Cette disposition étrange n'est pas sans raison d'être. Considérons le nombre décimal 2934, nous le lisons deux mille neuf cent trente quatre. La plus grande valeur, deux mille, est à gauche, tandis que la plus faible, quatre, est à droite. Nous voyons donc que la position de chaque chiffre dans un nombre est très significative puisqu'elle définit son *poids*.

La deuxième ligne du tableau 2.1 introduit une nouvelle représentation numérique. Chaque valeur de la base est suivie d'un petit nombre, appelé *exposant* qui correspond à la position dans le nombre. Par exemple, $1Ø^3$ se lit dix élevé à la puissance trois, et signifie 1Ø × 1Ø × 1Ø = 1ØØØ.

Tableau 2.1

Valeur	Milliers	Centaines	Dizaines	Unités
Représentation	$1Ø^3$	$1Ø^2$	$1Ø^1$	$1Ø^0$
Chiffre	2	9	3	4

Dans la représentation binaire d'un nombre, le poids de chaque bit est obtenu en élevant la base, 2, à la puissance x, x étant le numéro du bit dans l'octet (voir tableau 2.2). Par exemple, le bit numéro 7 a pour poids 2^7, c'est-à-dire :

2 × 2 × 2 × 2 × 2 × 2 × 2 = 128!

Tableau 2.2

Numéro du bit	7	6	5	4	3	2	1	Ø
Représentation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Poids	128	64	32	16	8	4	2	1

Conversion de nombres binaires en nombres décimaux

Comme il est possible de calculer le poids de chaque bit, il n'est pas très difficile de convertir les nombres binaires en nombres décimaux. Les règles de conversion sont :

Si le bit est à 1, on ajoute son poids.
Si le bit est à Ø, on l'ignore.

Essayons par exemple de convertir le nombre binaire, 1Ø1Ø1Ø1Ø, en nombre décimal.

$1 \times 128(2^7) = 128$
$\emptyset \times 64(2^6) = \emptyset$
$1 \times 32(2^5) = 32$
$\emptyset \times 16(2^4) = \emptyset$
$1 \times 8(2^3) = 8$
$\emptyset \times 4(2^2) = \emptyset$
$1 \times 2(2^1) = 2$
$\emptyset \times 1(2^0) = \emptyset$

17∅

Donc 1∅1∅1∅1∅ vaut 17∅ en base 1∅.

De même, 111∅111∅ représente :

$1 \times 128(2^7) = 128$
$1 \times 64(2^6) = 64$
$1 \times 32(2^5) = 32$
$\emptyset \times 16(2^4) = \emptyset$
$1 \times 8(2^3) = 8$
$1 \times 4(2^2) = 4$
$1 \times 2(2^1) = 2$
$\emptyset \times 1(2^0) = \emptyset$

238 en base 1∅.

Conversion de nombres décimaux en nombres binaires

Pour convertir un nombre décimal en un nombre binaire, les règles décrites ci-dessus sont simplement inversées : on soustrait successivement les poids binaires dans l'ordre décroissant. Si la soustraction est possible on place un 1 dans la colonne «binaire» et le reste est reporté sur la ligne suivante où le poids binaire suivant est retranché. Si la soustraction n'est pas possible, on place un ∅ dans la colonne «binaire» et le nombre est reporté à la ligne suivante. Par exemple, le tableau 2.3 montre la conversion en binaire du nombre décimal 141.

Tableau 2.3

Nombre décimal	Poids binaire	Binaire	Reste
141	$128(2^7)$	1	13
13	$64(2^6)$	∅	13
13	$32(2^5)$	∅	13
13	$16(2^4)$	∅	13
13	$8(2^3)$	1	5
5	$4(2^2)$	1	1
1	$2(2^1)$	∅	1
1	$1(2^0)$	1	∅

Donc 141 = 1∅∅∅11∅1 en binaire.

Conversion de nombres binaires en nombres hexadécimaux

Bien que la notation binaire soit probablement l'image la plus proche que nous puissions donner de la façon dont les nombres sont représentés à l'intérieur de l'Oric, vous avez sans doute déjà remarqué que les exemples de codes machine utilisent parfois des groupes de deux caractères précédés du signe « # ». Ces groupes sont des nombres hexadécimaux; leur valeur est calculée en base 16! Cela peut sembler peu maniable, mais en fait, nous allons voir que cette notation offre des avantages par rapport à la notation binaire ou décimale.

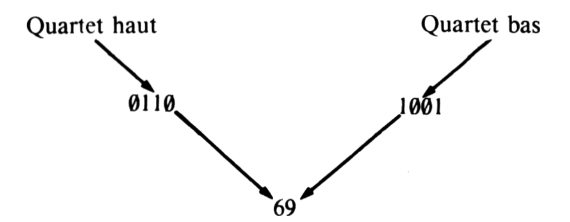
Il faut seize caractères différents pour représenter tous les chiffres possibles d'un nombre hexadécimal. Pour ce faire, on utilise les chiffres de Ø à 9 et les lettres A, B, C, D, E, F; ces dernières représentent les valeurs de 1Ø à 15. Le tableau 2.4 montre les équivalents binaires et décimaux des chiffres hexadécimaux.

Tableau 2.4

Décimal	Hexadécimal	Binaire
Ø	Ø	ØØØØ
1	1	ØØØ1
2	2	ØØ1Ø
3	3	ØØ11
4	4	Ø1ØØ
5	5	Ø1Ø1
6	6	Ø11Ø
7	7	Ø111
8	8	1ØØØ
9	9	1ØØ1
1Ø	A	1Ø1Ø
11	B	1Ø11
12	C	11ØØ
13	D	11Ø1
14	E	111Ø
15	F	1111

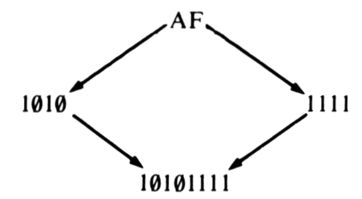
Pour convertir un nombre binaire en hexadécimal, on découpe d'abord l'octet en deux ensembles de quatre bits, appelés *quartets*, puis on utilise le tableau 2.4 pour trouver la valeur de chaque quartet.

Exemple, conversion de Ø11Ø 1ØØ1 en hexadécimal :



Il n'est pas toujours évident de distinguer un nombre hexadécimal d'un nombre décimal (comme dans l'exemple ci-dessus). De ce fait, pour l'Oric, on convient de faire précéder les nombres hexadécimaux du signe « # ». Ainsi Ø11Ø1ØØ1 s'écrit #69 en hexadécimal. En inversant le procédé, les nombres hexadécimaux peuvent facilement être convertis en binaire.

Exemple, conversion de #AF en binaire :



Il est maintenant clair qu'il est plus facile de convertir des nombres hexadécimaux en nombres binaires (et vice versa) que de convertir des nombres décimaux en nombres binaires. On remarquera que le plus grand nombre binaire qu'il soit possible de représenter par un octet, 11111111, ne s'écrit qu'avec deux chiffres hexadécimaux : #FF.

Conversion de nombres hexadécimaux en nombres décimaux

Voyons enfin comment convertir un nombre hexadécimal en un nombre décimal. Pour cela on fait la somme des produits de la valeur décimale de chaque chiffre par son poids.

Exemple, conversion de #31A en décimal :

Le 3 représente $3 \times 16^2 = 3 \times 16 \times 16 = 768$
Le 1 représente $1 \times 16^1 = 1 \times 16 = 16$
Le A représente $1Ø \times 16^0 = 1Ø \times 1 = 1Ø$

En additionnant le tout, on voit que #31A = 794 en décimal.

La conversion d'un nombre décimal en hexadécimal est un peu plus compliquée. Il faut diviser le nombre décimal plusieurs fois par 16 jusqu'à ce que l'on obtienne une valeur plus petite que 16. Le chiffre hexadécimal correspondant à cette valeur est noté et on recommence la même opération sur le reste, tant qu'il est supérieur à 16.

Exemple, conversion de 4Ø72 en hexadécimal :

4Ø72 ÷ 16 ÷ 16 = 15 = F (reste = 4Ø72 − (15 × 16 × 16) = 232)
232 ÷ 16 = 14 = E (reste = 232 − (14 × 16) = 8)
8 = 8

Donc 4Ø72 = #FE8 en hexadécimal.

Ces deux conversions sont un peu tortueuses (c'est le moins qu'on puisse dire!), mais après tout, comme nous disposons d'un micro-ordinateur très puissant, laissons-le faire ce travail fastidieux!

Pour afficher la valeur décimale d'un nombre hexadécimal tel que #31A, tapez :

PRINT #31A

et pour afficher la valeur hexadécimale d'un nombre décimal, faites :

PRINT HEX$(4Ø72)

Chapitre 3

Le compte est bon !

L'arithmétique binaire

Je vous en prie, ne vous laissez pas impressionner et ne sautez pas ce chapitre, simplement parce que vous y avez vu ce mot terrifiant : «arithmétique». L'addition et la soustraction de nombres binaires sont des opérations simples; en fait il vous suffit de savoir compter jusqu'à deux! Bien qu'il ne soit pas indispensable de savoir soustraire et additionner des Ø et des 1 «à la main», ce chapitre va vous familiarisez avec plusieurs concepts nouveaux et vous aidera à mieux comprendre les chapitres suivants.

L'addition

Il n'existe que quatre règles simples pour additionner des nombres binaires :

1. Ø + Ø = Ø
2. 1 + Ø = 1
3. Ø + 1 = 1
4. 1 + 1 = (1)Ø

Notez que dans la règle 4 le résultat de 1 + 1 est (1)Ø. Le 1 entre parenthèses s'appelle la retenue, elle indique qu'il y a un débordement d'une colonne sur une autre; souvenez-vous, 1Ø en binaire est égal à 2 en décimal. Cette retenue ressemble à celle qu'on obtient en additionnant deux chiffres décimaux lorsque le résultat est supérieur à 9. Par exemple, si on ajoute 9 et 1, on obtient 1Ø, ce qui s'écrit en plaçant le Ø dans la colonne des unités et en faisant «déborder» la retenue dans la colonne suivante. De même, dans une addition binaire, quand le résultat est supérieur à 1, on ajoute la retenue à la colonne suivante. Essayons d'appliquer ces principes et additionnons deux nombres binaires de quatre chiffres, Ø1Ø1 et Ø1ØØ.

```
 Ø1Ø1        (#5)
+Ø1ØØ        (#4)
-----
 1ØØ1        (#9)
```

En lisant chaque colonne à partir de la gauche on obtient :

Première colonne :	1 + Ø	= 1
Deuxième colonne :	Ø + Ø	= Ø
Troisième colonne :	1 + 1	= Ø(1)
Quatrième colonne :	Ø + Ø = Ø + (1)	= 1

Dans cet exemple la troisième colonne a produit une retenue qui a été ajoutée à la quatrième colonne. On procède de la même manière pour additionner des nombres de huit chiffres :

```
 Ø1Ø1Ø1Ø1        (#55)
+Ø111ØØ1Ø        (#72)
---------
 11ØØØ111        (#C7)
```

La soustraction

Jusqu'à présent, nous ne nous sommes occupés que de nombres positifs. Mais pour soustraire des nombres binaires, il faut que nous soyons capables de représenter aussi bien des nombres positifs que négatifs. Dans une soustraction binaire nous allons utiliser une technique légèrement différente de celle qu'on utilise d'habitude; en fait nous ne ferons pas de soustraction du tout : nous allons additionner l'opposé du nombre qu'il faut soustraire. Par exemple, au lieu de faire 4 − 3 (quatre moins trois) nous faisons 4 + (− 3) (quatre plus moins trois!). J'espère que la figure 3.1 va supprimer la confusion et les migraines que tout ceci provoque!

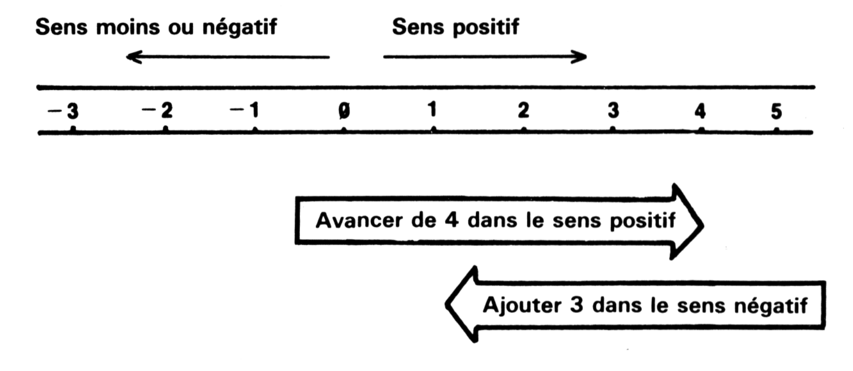
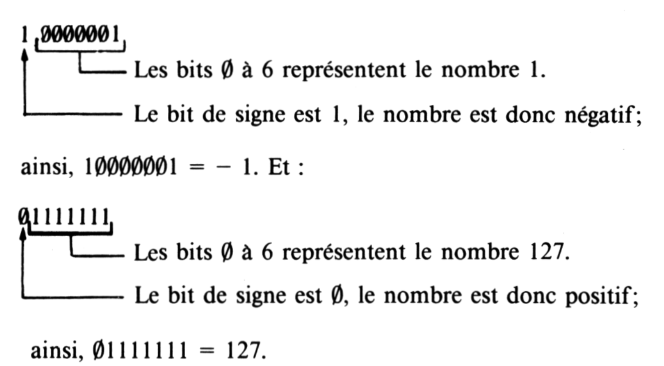


Figure 3.1 Diagramme de l'opération 4 + (− 3).

Nous pouvons nous servir de cette graduation pour résoudre 4 + (− 3). Le point de départ est Ø. Tout d'abord on avance jusqu'au point 4 (c'est-à-dire 4 pas dans le sens positif) et on ajoute − 3 (c'est-à-dire 3 pas dans le sens négatif). Nous nous

trouvons maintenant au point 1, là où nous devions être. Essayez d'utiliser cette méthode pour soustraire 8 de 12, pour être sûr d'avoir compris.

Voyons maintenant comment cela s'applique aux nombres binaires. Mais d'abord, comment représente-t-on un nombre négatif en binaire? On emploie une représentation appelée «binaire-signé» où le bit 7 de l'octet, appelé le bit de poids fort, marque le signe du nombre. Par convention, le bit 7 vaut Ø pour un nombre positif et 1 pour un nombre négatif. Par exemple, en binaire-signé :



ainsi, 1ØØØØØØ1 = − 1. Et :

ainsi, Ø1111111 = 127.

Cependant, il ne suffit pas d'indiquer le signe avec le bit 7 pour obtenir une représentation efficace des nombres binaires négatifs. En fait, on change le signe d'un nombre binaire en prenant son *complément à deux.* Pour cela on inverse chaque bit du nombre puis on lui ajoute 1. Pour représenter − 3, commençons par écrire 3 en binaire :

Ø Ø Ø Ø Ø Ø 1 1

Maintenant inversons chaque bit. (On remplace chaque Ø par un 1 et chaque 1 par un Ø : cela s'appelle le *complément à un*).

1 1 1 1 1 1 Ø Ø

ajoutons 1 :

```
  1 1 1 1 1 1 Ø Ø
+               1
  ---------------
  1 1 1 1 1 1 Ø 1
```

Ainsi la représentation de − 3 en complément à deux est 111111Ø1.

Utilisons maintenant ce résultat pour faire l'opération 4 + (− 3) :

```
(4)            Ø Ø Ø Ø Ø 1 Ø Ø
(− 3)          1 1 1 1 1 1 Ø 1
               ---------------
(Somme)    (1) Ø Ø Ø Ø Ø Ø Ø 1
```

Nous voyons que le résultat est bien 1, mais le débordement du bit 7 a produit une retenue. Pour l'instant nous n'en tiendrons pas compte, mais nous verrons plus loin qu'elle peut être importante.

Un exemple de plus n'est peut-être pas inutile. Faisons 32 – 16, c'est-à-dire 32 + (– 16).

32 s'écrit en binaire :

Ø Ø 1 Ø Ø Ø Ø Ø

16 s'écrit en binaire :

Ø Ø Ø 1 Ø Ø Ø Ø

Le complément à deux de 16 est :

```
  1 1 1 Ø 1 1 1 1
+               1
  ---------------
  1 1 1 1 Ø Ø Ø Ø
```

Maintenant additionnons :

```
(32)             Ø Ø 1 Ø Ø Ø Ø Ø
(– 16)         + 1 1 1 1 Ø Ø Ø Ø
                 ---------------
(16)          (1) Ø Ø Ø 1 Ø Ø Ø Ø
```

Si nous ne tenons pas compte de la retenue, nous avons notre résultat : 16.

Ces exemples nous montrent qu'avec les règles de l'addition binaire il est possible d'additionner ou de soustraire des entiers relatifs. La retenue mise à part, le résultat est juste et le signe est correct. Il est également possible d'additionner deux nombres négatifs et d'obtenir un résultat négatif correct. Effectuons (– 2) + (– 2) en utilisant les compléments à deux.

2 s'écrit en binaire :

Ø Ø Ø Ø Ø Ø 1 Ø

Le complément à deux est :

```
  1 1 1 1 1 1 Ø 1
+               1
  ---------------
  1 1 1 1 1 1 1 Ø
```

Nous pouvons ajouter ce nombre à lui-même pour obtenir :

```
(– 2)            1 1 1 1 1 1 1 Ø
(– 2)          + 1 1 1 1 1 1 1 Ø
                 ---------------
              (1) 1 1 1 1 1 1 Ø Ø
```

Si l'on ne tient pas compte de la retenue, le résultat est bien – 4. Vous pouvez vérifier en prenant le complément à deux de – 4 par la méthode habituelle.

Les opérations logiques

La logique théorique prend en compte les situations dans lesquelles il n'y a que deux possibilités, vrai ou faux. En termes binaires, ces deux possibilités sont représentées par 1 et Ø. On peut effectuer sur des nombres binaires trois opérations logiques différentes : AND (et), OR (ou) et EOR (ou exclusif). Chaque opération s'effectue entre les bits de même rang de deux nombres.

AND

Les quatre règles pour AND sont :

1. Ø AND Ø = Ø
2. 1 AND Ø = Ø
3. Ø AND 1 = Ø
4. 1 AND 1 = 1

Comme on le voit clairement, l'opération AND ne peut produire 1 que si les deux bits testés valent tous les deux 1. Si au moins l'un des bits vaut Ø, le résultat sera Ø.

Exemple, appliquons la fonction AND aux deux nombres suivants :

```
      1 Ø 1 Ø
AND   Ø Ø 1 1
      -------
      Ø Ø 1 Ø
```

Dans le résultat, seul le bit 1 est à 1, tous les autres sont à Ø puisque, dans chaque cas, un au moins des bits testés vaut Ø. On utilise AND principalement pour «masquer» ou «conserver» certains bits. Imaginons que nous voulions protéger les quatre bits de poids faible d'un octet (quartet bas), et mettre à zéro les quatre bits de poids fort (quartet haut). Il faut pour cela appliquer la fonction AND à l'octet et au nombre ØØØØ1111. Si l'octet contient 1Ø1Ø11ØØ, le résultat sera :

```
      1 Ø 1 Ø 1 1 Ø Ø        (octet testé)
AND   Ø Ø Ø Ø 1 1 1 1        (masque)
      ---------------
      Ø Ø Ø Ø 1 1 Ø Ø
```

Le quartet haut est mis à zéro et le quartet bas est conservé !

OR

Les quatre règles pour OR sont :

1. Ø OR Ø = Ø
2. 1 OR Ø = 1
3. Ø OR 1 = 1
4. 1 OR 1 = 1

La fonction OR a pour résultat 1 si l'un au moins des bits testés vaut 1. On ne peut obtenir Ø que si aucun des bits ne vaut 1.

Exemple, appliquons la fonction OR aux deux nombres suivants :

```
      1 Ø 1 Ø
OR    Ø Ø 1 1
      -------
      1 Ø 1 1
```

Seul le bit 2 est à zéro, les autres bits sont tous à 1, puisque chaque paire testée contient au moins un 1. On utilise communément la fonction OR pour mettre à 1 certains bits (on dit «forçer les bits»). Par exemple, si vous voulez forçer les bits Ø et 7 d'un octet, vous appliquerez la fonction OR à l'octet et au nombre 1ØØØØØØ1.

```
      Ø Ø 1 1 Ø 1 1 Ø        (octet testé)
OR    1 Ø Ø Ø Ø Ø Ø 1        (octet de forçage)
      ---------------
      1 Ø 1 1 Ø 1 1 1
```

Les bits Ø et 7 sont forcés à 1, tandis que les autres ne sont pas modifiés.

EOR

Comme AND et OR, EOR obéit à quatre règles :

1. Ø EOR Ø = Ø
2. 1 EOR Ø = 1
3. Ø EOR 1 = 1
4. 1 EOR 1 = Ø

Cette fonction est un OU exclusif, en d'autres termes si les deux bits testés ont la même valeur le résultat est Ø. On ne peut obtenir 1 que si les deux bits ont des valeurs différentes.

Exemple, appliquons la fonction EOR aux deux nombres suivants :

```
      1 Ø 1 Ø
EOR   Ø Ø 1 1
      -------
      1 Ø Ø 1
```

Cette opération est souvent utilisée pour inverser les bits d'un octet (complément à un). Essayons en appliquant la fonction EOR à un octet et au nombre 11111111.

```
      1 Ø Ø 1 1 Ø Ø Ø        (octet à inverser ou à complémenter)
EOR   1 1 1 1 1 1 1 1        (octet d'inversion)
      ---------------
      Ø 1 1 Ø Ø 1 1 1
```

Comparons le résultat et l'octet de départ : ils sont bien exactement inverses.

Chapitre 4

Les registres

Pour réaliser ses différentes tâches, le 65Ø2 utilise plusieurs emplacements spéciaux, appelés *registres*. Comme ces registres sont internes au 65Ø2, ils ne figurent pas dans la carte de la mémoire de l'Oric (voir annexe 6). Ces registres ne sont désignés que par leur nom. La figure 4.1 montre les registres du 65Ø2. Pour l'instant nous ne nous occuperons que de quatre d'entre eux, c'est-à-dire l'accumulateur, les registres X et Y et le compteur de programme.

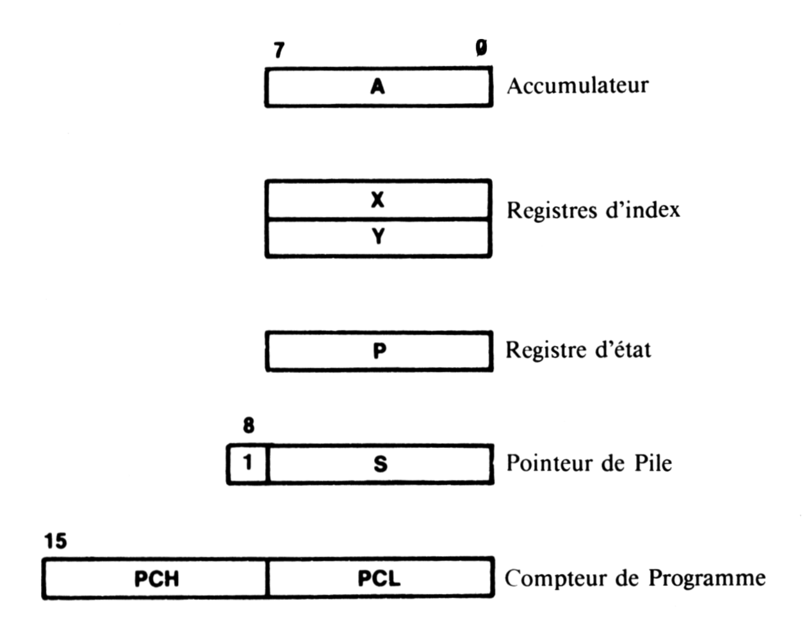


Figure 4.1 Les registres du 65Ø2.

L'accumulateur

Nous avons déjà parlé plusieurs fois de l'accumulateur (registre «A») dans les chapitres précédents. Comme vous l'avez deviné, l'accumulateur est le registre principal du 65Ø2, et comme la plupart des autres, c'est un registre de 8 bits. Ce qui signifie qu'il ne peut contenir qu'un seul octet d'information à un instant donné. En tant que registre principal, la plupart des instructions y font référence, et sa spécificité est due au rôle qu'il joue dans l'exécution des opérations logiques et arithmétiques.

Le tableau 4.1 présente les instructions liées à l'accumulateur. Il n'est pas indispensable de les retenir dans l'immédiat, mais elles sont introduites dès à présent pour que vous vous familiarisiez avec elles.

Tableau 4.1

Les instructions liées à l'accumulateur			
ADC	Addition avec retenue	ROL	Rotation à gauche
AND	ET logique	ROR	Rotation à droite
ASL	Décalage arithmétique à gauche	SBC	Soustraction avec retenue
BIT	Test de bits	STA	Stockage de l'accumulateur
CMP	Comparaison avec l'accumulateur	TAX	Transfert de l'accumulateur dans le registre X
EOR	OU exclusif logique	TAY	Transfert de l'accumulateur dans le registre Y
LDA	Chargement de l'accumulateur	TXA	Transfert du registre X dans l'accumulateur
LSR	Décalage logique à droite	TYA	Transfert du registre Y dans l'accumulateur
ORA	OU logique		
PHA	Empilement de l'accumulateur		
PLA	Dépilement de l'accumulateur		

Les registres d'index

Il y a deux autres registres dans le 65Ø2 qui ne peuvent contenir qu'un seul octet de données. Ce sont le *registre X* et le *registre Y*. Ils sont généralement appelés «registres d'index» car ils sont souvent utilisés pour indiquer un déplacement (index) par rapport à une adresse de base spécifiée par ailleurs. Comme il y a des instructions qui permettent de les incrémenter et de les décrémenter directement (ce qui n'est pas le cas pour l'accumulateur), ils sont souvent utilisés comme compteurs. S'il n'est pas possible de réaliser des opérations arithmétiques ou logiques dans ces registres d'index, on peut toutefois transférer leur contenu dans l'accumulateur et vice versa.

Les instructions liées à ces deux registres sont données dans le tableau 4.2.

Tableau 4.2

Les instructions liées au registre X		Les instructions liées au registre Y	
CPX	Comparaison avec le registre X	CPY	Comparaison avec le registre Y
DEX	Décrémentation du registre X	DEY	Décrémentation du registre Y
INX	Incrémentation du registre X	INY	Incrémentation du registre Y
LDX	Chargement du registre X	LDY	Chargement du registre Y
STX	Stockage du registre X	STY	Stockage du registre Y
TAX	Transfert de l'accumulateur dans le registre X	TAY	Transfert de l'accumulateur dans le registre Y
TXA	Transfert du registre X dans l'accumulateur	TYA	Transfert du registre Y dans l'accumulateur
TSX	Transfert du registre d'état dans le registre X		
TXS	Transfert du registre X dans le registre d'état		

Le compteur de programme

Le compteur de programme est le carnet d'adresses du 65Ø2. Il contient presque toujours l'adresse en mémoire de la prochaine instruction à exécuter. Contrairement aux autres registres, c'est un registres de 16 bits, formé de deux registres de 8 bits. Ces deux derniers sont souvent appelés le compteur de programme haut (PCH) et le compteur de programme bas (PCL).

Chapitre 5

Un essai en code machine

Après avoir vu quelques uns des principes de base, pourquoi ne pas écrire notre premier programme en code machine ? Après tout, c'est l'objet de ce livre !

Comme vous le savez sans doute, on peut utiliser REM pour inclure dans un programme des commentaires destinés à l'utilisateur, que le programme ignorera. Dans les listings ci-dessous, REM suivi de *deux astérisques* sert à indiquer un titre de programme ou de sous-programme, et REM suivi d'*un astérisque* est utilisé pour expliquer ce que fait le code machine. Chaque ligne de code machine sous forme de DATA contient également les mnémoniques correspondants (après REM sans astérisque). Entrez le programme suivant, en omettant éventuellement les commentaires.

Programme 1

```
10  REM * * Exemple de code machine * *
20  REM * * Affiche "A" à l'écran * *
30  CODE = #400
40  FOR INDEX = 0 TO 5
50    READ OCTET
60    POKE CODE + INDEX, OCTET
70  NEXT INDEX
80
90  REM * * Les données du code machine * *
95  REM * Charge le code de "A" dans l'accumulateur *
100 DATA #A9, #41       : REM LDA @ASC"A"
105 REM * Range le contenu de l'accumulateur en #BF50 *
110 DATA #8D, #50, #BF : REM STA #BF50
115 REM * Retour sous Basic *
```

```
120 DATA #60               : REM RTS
130
140 REM * * Exécution du code machine * *
150 CLS
160 CALL (CODE)
```

Ce programme affiche la lettre «A» à l'écran. Ce n'est rien de spectaculaire, mais il fait intervenir divers éléments que vous retrouverez dans tous vos programmes en code machine. Voici ce que signifie chaque ligne :

Ligne 30	Déclaration d'une variable appelée CODE pour indiquer où sera rangé le code machine.
Ligne 40	Initialise une boucle de lecture de données.
Ligne 50	Lecture d'un *octet* de code machine.
Ligne 60	Ecriture de cet octet en mémoire.
Ligne 70	Répétition de la boucle jusqu'à ce que ce soit fini.
Ligne 100	Données du code machine.
Ligne 110	Données du code machine.
Ligne 120	Données du code machine.
Ligne 150	Effacement de l'écran.
Ligne 160	Exécution du code machine.

Pour voir ce que fait ce programme, tapez RUN, actionnez la touche RETURN, et voilà : vous voyez apparaître un «A» sur l'écran.

Essayons de répondre à deux questions qui viennent immédiatement à l'esprit, à savoir :

1. Où peut-on ranger le code machine ?
2. Comment peut-on entrer du code machine ?

CODE - le compteur de programme

Il apparaît de façon évidente que le code machine que nous écrivons doit être rangé quelque part en mémoire. Dans tous les programmes de ce livre, j'ai utilisé la variable BASIC «CODE» comme un pointeur vers l'adresse en mémoire à partir de laquelle le code machine doit être rangé. (En fait, CODE agit plutôt comme le compteur de programme du microprocesseur). Vous avez peut-être envie d'utiliser un autre nom de variable, cela n'a aucune importance. Vous pouvez penser que DC ou même CODEMACHINE sont des noms plus appropriés pour marquer le début du code. L'essentiel est de prendre l'habitude d'utiliser le même nom de variable dans tous vos programmes pour éviter des ambiguïtés.

Il faut choisir avec soin la valeur donnée à CODE. Un choix inconsidéré peut facilement détruire un autre programme, voire même votre propre programme en code machine! Dans le programme 1, CODE est initialisée à #400 avec l'instruction d'affectation habituelle :

```
CODE = #400
```

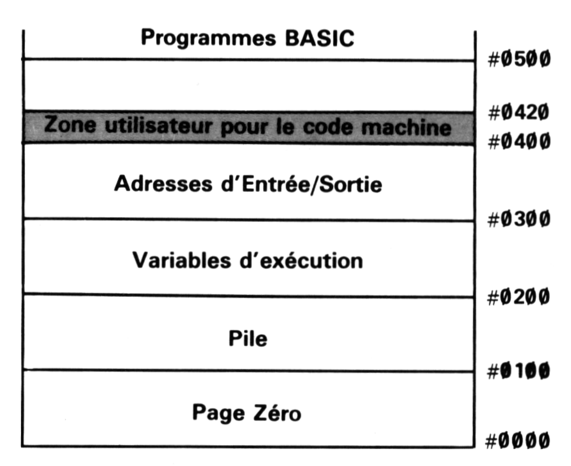


Figure 5.1 La zone utilisateur pour le code machine

et les 6 octets du code machine sont rangés ici, ou plus exactement dans les 6 octets à partir de #4ØØ (#4ØØ à #4Ø5). Si vous regardez la figure 5.1, vous verrez que cette zone a été spécialement réservée aux programmes en code machine. On dispose à cet endroit de la mémoire de 32 octets (#4ØØ à #42Ø).Il ne devrait donc pas y avoir de problèmes si vous utilisez cette zone, à condition, bien sûr, que vos programmes ne nécessitent pas plus de 32 octets.

Et pourtant il arrive fréquemment que des programmes en code machine soient plus longs. Il faut donc chercher une autre possibilité pour les ranger en mémoire. On peut envisager deux solutions.

Premièrement, si vous n'utilisez que le mode TEXT, vous pouvez vous servir de la zone en mémoire normalement réservée au mode HIRES. Un coup d'oeil à la figure 5.2 vous apprendra que cette zone est comprises entre #98ØØ et #B4ØØ, soit un bloc de 7168 octets, ou plus simplement, de 7K. Cela est amplement suffisant, même pour les programmes en code machine très longs. En revanche, il sera dangereux d'utiliser la commande GRAB après avoir rangé votre code à cet endroit-là.

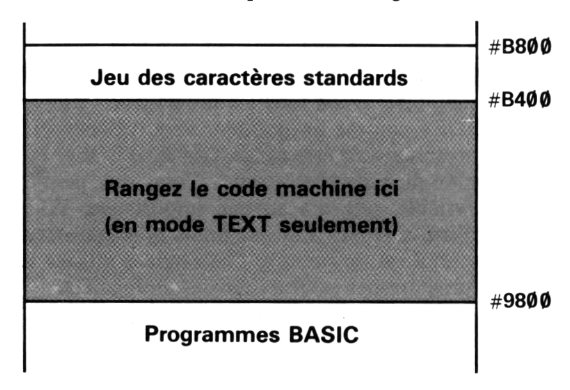


Figure 5.2 Rangement du code machine dans la zone HIRES

La méthode la meilleure et la plus sûre est peut-être de déplacer vers le bas la limite supérieure de la zone réservée aux programmes BASIC et de ranger le code

machine au-dessus de cette zone. Cela permet d'éviter qu'il soit détruit, puisque les programmes BASIC ne peuvent dépasser la limite de la zone qui leur est réservée, et le rend indépendant des zones associées aux modes TEXT et HIRES.

HIMEM (abréviation pour HIgh MEMory point, en anglais) est une pseudo-variable associée à la limite supérieure de la zone des programmes BASIC. On peut donner une valeur à cette variable, mais on ne peut la lire directement. Il existe quatre octets au début de la carte de la mémoire de l'Oric (dans ce qu'on appelle la Page Zéro, dont nous aurons à reparler) dont on peut penser qu'ils sont associés à HIMEM, bien qu'on ne sache pas exactement de quelle façon. Ce sont :

#A2 et #A3
#A6 et #A7

La figure 5.3 montre que la limite supérieure de la zone des programmes est en #98ØØ. Tout ce que nous avons à faire est de calculer le nombre d'octets nécessaires pour notre code machine, d'en ajouter quelques uns par mesure de sécurité, et de réinitialiser HIMEM. Supposons par exemple, que nous ayons besoin d'à peu près 5ØØ octets. Après avoir converti ce nombre en hexadécimal et l'avoir retranché de #98ØØ, on arrive en gros à #96ØØ. Pour initialiser HIMEM avec cette valeur tapez :

HIMEM #96ØØ

Remarquez qu'on n'a pas besoin du signe «=» et qu'il suffit d'un espace. Cela fait, seul GRAB peut modifier sa valeur. Utiliser NEW ou reaffecter HIMEM est sans effet.

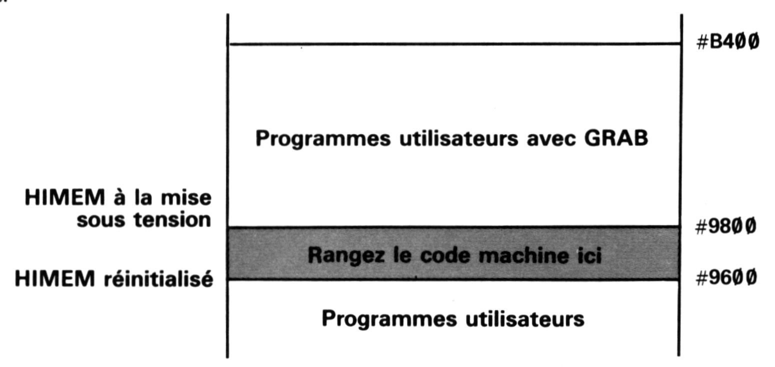


Figure 5.3 Rangement du code machine au-dessus de HIMEM

De plus, l'utilisation de ces emplacements au-dessus de HIMEM présente l'avantage de libérer la zone réservée au code machine. Vous pourrez vous en servir dans vos programmes comme d'une zone de rangement de données ou d'adresses.

En résumé, on peut considérer les zones suivantes commes raisonnablement «sûres» pour y ranger du code machine :

1. La zone utilisateur réservée au code machine (#4ØØ à #42Ø).
2. La partie de la mémoire normalement associée au mode HIRES (#98ØØ à #B4ØØ).
3. Au-dessus de HIMEM après réinitialisation.

L'entrée du code machine

Le programme 1 est un exemple de la méthode la plus évidente qui permet d'entrer du code machine. Les nombres hexadécimaux sont écrits sous forme d'une série de DATA, qui est lue à l'aide d'une boucle FOR... NEXT et écrite en mémoire avec l'instruction POKE et un compteur de boucle qui sert d'INDEX à partir de l'adresse de base (donnée par CODE). Ceci permet d'écrire les octets dans des emplacements successifs en mémoire.

Remarquez également dans ce programme 1, comme dans toute la suite de ce livre, que chaque ligne de DATA est consacrée à une seule instruction en code machine, suivie d'un commentaire rappelant le mnémonique correspondant au code opérateur de l'instruction. Par exemple :

```
1ØØ  DATA #A9, #41 : REM LDA @ASC"A"
```

Il serait bon de prendre cette habitude, tout simplement parce que cela facilite la relecture de vos programmes. C'est particulièrement important dans le cas où vous voulez revenir à un programme que vous avez écrit plusieurs semaines auparavant, dont vous avez presque certainement oublié les détails. Une ligne de DATA ne contenant que des nombres hexadécimaux ne vous aidera pas à retrouver la mémoire :

```
123  DATA #A9, #7Ø, #85, #22, #8D, #5Ø, #9Ø, #6Ø
```

si vous voyez ce que je veux dire!

Un dernier point concernant le compteur de la boucle. Lorsque vous l'initialisez, comptez le nombre total d'octets, moins un. (Rappelez-vous que le compteur de boucle doit commencer à Ø, pour que le premier octet soit réellement écrit à l'adresse spécifiée par CODE — CODE + INDEX = #4ØØ + Ø = #4ØØ.)

Bien que la méthode ci-dessus soit probablement la meilleure façon d'entrer du code machine, elle est un peu compliquée. Il est plus facile d'utiliser un chargeur hexadécimal. C'est un programme qui vous demande de taper le code machine sous forme de nombres hexadécimaux et qui les écrit en mémoire. Le programme deux est un exemple simple, mais efficace d'un tel chargeur.

Programme 2

```
1Ø  REM * * Chargeur hexadécimal pour l'ORIC * *
2Ø  CLS
3Ø  INK 3 : PAPER 4
4Ø  PRINT "Chargeur de l'ORIC" : PRINT
5Ø  INPUT "Adresse d'implantation :"; A$
6Ø  ADR = VAL(A$)
7Ø  REPEAT
8Ø    PRINT HEX$(ADR); " : #";
9Ø    GET B$
```

```
100    IF B$ = "S" GOTO 250 : REM * STOP *
110    IF ASC(B$) > ASC("F") GOTO 90
120    B$ = "#" + B$ : HAUT = VAL(B$)
130    IF HAUT > 15 GOTO 90
140    IF HAUT < 0 GOTO 90
150    PRINT RIGHT$(B$, 1);
160    GET C$
170    IF ASC(C$) > ASC("F") GOTO 160
180    C$ = "#" + C$ :BAS = VAL(C$)
190    IF BAS > 15 GOTO 160
200    IF BAS < 0 GOTO 160
210    PRINT RIGHT$(C$, 1)
220    OCTET = HAUT * 16 + BAS
230    POKE ADR, OCTET
240    ADR = ADR + 1
250  UNTIL B$ = "S"
```

Entrez et exécutez ce programme. Après avoir affiché l'en-tête, il vous demande de taper l'adresse d'implantation. Il s'agit de l'adresse à partir de laquelle vous voulez ranger le code machine en mémoire. C'est la valeur que vous auriez donnée à CODE. Vous pouvez la taper sous forme décimale, ou sous forme hexadécimale, à condition de la faire précéder du signe «#». Appuyez sur la touche RETURN et vous verrez apparaître la première adresse codée en hexadécimal suivie par un autre «#». Il vous suffit maintenant de taper les deux chiffres hexadécimaux (sans le dièse qui est déjà là). Dès que vous avez tapé le deuxième chiffre, l'octet est écrit en mémoire à l'emplacement indiqué et l'adresse suivante est affichée en dessous.

Pour sortir du programme, tapez «S» (pour Stop) à la suite de la dernière adresse affichée. Ce programme teste également les chiffres que vous lui donnez et rejette les caractères qui ne sont pas des chiffres hexadécimaux. La figure 5.4 montre ce qu'affiche ce programme à l'écran.

```
Chargeur pour l'ORIC

Adresse d'implantation : ? #400
#400  :  #A9
#401  :  #41
#402  :  #8D
#403  :  #50
#404  :  #BF
#405  :  #60
#406  :  #S

Ready
```

Figure 5.4 Le chargeur hexadécimal.

L'exécution du code machine

Pour exécuter un programme en code machine, on utilise l'instruction CALL du BASIC, pour signaler à l'interpréteur du BASIC où est situé le code machine. L'instruction CALL doit être suivie d'une étiquette ou d'une adresse. Ainsi, pour exécuter le code machine généré par le programme d'assemblage, vous pouvez taper soit :

CALL(CODE)

qui est l'étiquette marquant la première adresse du programme, soit :

CALL(#4ØØ)

qui est cette première adresse.

L'enregistrement du code machine

Il serait bon de prendre l'habitude de ranger vos programmes en code machine sur cassettes AVANT de les faire tourner. Cela peut vous sembler curieux, puisque c'est l'inverse de ce que vous faites en BASIC, où vous n'enregistrez un programme qu'après l'avoir testé et corrigé. Mais un programme en code machine que l'on fait tourner pour la première fois, risque de contenir une erreur qui peut bloquer votre ORIC. La seule solution sera alors de l'éteindre et de recommencer le tout. RESET ne sert à rien dans ce cas. Si cela vous arrive, mais que vous avez déjà enregistré votre programme sur cassette, il ne vous reste plus qu'à le recharger et à corriger l'erreur!

Vous trouverez tous les détails sur les transferts entre la mémoire et une cassette dans le chapitre 11 du *Manuel.*

Chapitre 6

Les indicateurs d'état

Le registre d'état

Le registre d'état est différent des autres registres du 65Ø2. Quand nous l'utilisons, ce n'est pas la valeur hexadécimale qu'il contient qui nous intéresse, mais plutôt l'état de ses différents bits. Chaque bit fournit une indication sur les conditions dans lesquelles se déroule le programme. Seuls sept des huit bits qui composent ce registre ont une signification (le bit 5 est toujours à 1). La figure 6.1 montre la place de ces différents indicateurs, et chacun d'eux est ensuite décrit en détail.

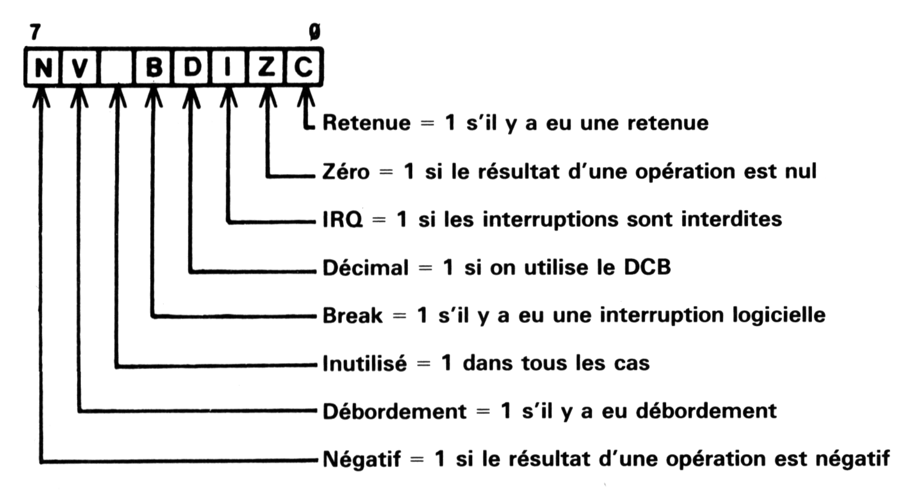


Figure 6.1 Les indicateurs du registre d'état.

Bit 7 : l'indicateur Négatif (N)

En binaire signé, l'indicateur négatif est utilisé pour indiquer le signe d'un nombre. Si cet indicateur vaut 1 (N = 1), ce nombre est négatif. Si cet indicateur vaut Ø (N = Ø), le nombre est positif. Toute une série d'instructions affectent cet indicateur, en particulier les instructions logiques et arithmétiques. En général, le bit le plus significatif du résultat d'une opération est directement recopié dans l'indicateur N. Considérons les deux opérations suivantes :

```
LDA @#8Ø          \ charger l'accumulateur avec #8Ø
```

Cette instruction aura pour effet de mettre à 1 l'indicateur Négatif (N = 1) car #8Ø = 1ØØØØØØØ en binaire. Par contre :

```
LDA @#7F          \ charger l'accumulateur avec #7F
```

mettra l'indicateur Négatif à Ø, car #7F = Ø1111111 en binaire. Il y a deux instructions qui utilisent l'indicateur Négatif, à savoir :

BMI Brancher si moins (N = 1)
BPL Brancher si plus (N = Ø)

On reviendra sur tout cela plus tard.

Bit 6 : l'indicateur de Débordement (V)

Cet indicateur est probablement le moins utilisé de tous les indicateurs du registre d'état. Il indique s'il y a eu un débordement au niveau du bit 6 pendant une addition ou une soustraction. Dans ces deux cas, l'indicateur de Débordement est mis à 1 (V = 1).

Examinons les exemples suivants : d'abord, #Ø9 + #Ø7 :

```
(#Ø9)        Ø Ø Ø Ø 1 Ø Ø 1
(#Ø7)      + Ø Ø Ø Ø Ø 1 1 1
             ---------------
(#1Ø)        Ø Ø Ø 1 Ø Ø Ø Ø
             ↑
             └── Pas de débordement au niveau du bit 6, donc V = Ø
```

Deuxièmement, #7F + #Ø1 :

```
(#7F)        Ø 1 1 1 1 1 1 1
(#Ø1)      + Ø Ø Ø Ø Ø Ø Ø 1
             ---------------
(#8Ø)        1 Ø Ø Ø Ø Ø Ø Ø
             ↑
             └── Il y a eu débordement au niveau du bit 6, donc V = 1
```

Si nous avions utilisé le binaire signé, nous aurions obtenu −128, ce qui est bien sûr un résultat incorrect. Comme le débordement est indiqué, nous pouvons corriger le résultat.

Bit 5

Ce bit n'a pas de signification, il est toujours à 1.

Bit 4 : l'indicateur d'interruption logicielle (B)

Cet indicateur est mis à 1 si une interruption logicielle (BREAK) survient, autrement il est à Ø. Cela peut vous sembler farfelu, puisqu'il vous sera facile de voir si votre programme s'interrompt. Mais cette interruption peut être provoquée par des causes externes. Cet indicateur fait justement la différence entre le BREAK et les autres interruptions.

Bit 3 : l'indicateur Décimal (D)

Il indique au processeur le type d'arithmétique qui est utilisé. S'il est mis à Ø (par l'instruction CLD), les opérations se font en hexadécimal, ce qui est l'état usuel, S'il est mis à 1 (par l'instruction SED), tous les nombres seront considérés comme codés en DCB.

Bit 2 : l'indicateur d'Interruption IRQ (I)

Nous avons déjà parlé d'interruptions à propos de l'indicateur B. Pour l'instant, il nous suffit de savoir que cet indicateur vaut 1 quand l'interruption IRQ est interdite et qu'il vaut Ø quand les interruptions IRQ sont autorisées.

Bit 1 : l'indicateur Zéro (Z)

Celui-ci indique si un résultat est nul ou pas. S'il est nul, l'indicateur est mis à 1 (Z = 1), autrement il vaut Ø (Z = Ø). On peut dire que l'indicateur Zéro est affecté par les mêmes instructions que l'indicateur Négatif. Regardons :

```
LDA @Ø         \ charger l'accumulateur avec zéro
```

Cette instruction mettra l'indicateur Zéro à 1 (Z = 1), mais :

```
LDX @#FA    \ charger le registre X avec #FA
```

mettra l'indicateur à Ø (Z = Ø).

Bit Ø : l'indicateur de Retenue (C)

Nous avons déjà vu qu'au cours d'une addition de deux octets, il peut se produire une retenue, qu'on reporte alors d'un bit sur un autre. Que se passe-t-il lorsque la retenue se produit sur le bit le plus significatif?

Par exemple, quand nous ajoutons #FF à #8Ø :

```
(#FF)              1 1 1 1 1 1 1 1
(#8Ø)         +    1 Ø Ø Ø Ø Ø Ø Ø
                  ------------------
(#7F)            (1) Ø 1 1 1 1 1 1 1
                  └── Retenue provenant du bit 7
```

Le résultat est trop grand pour tenir sur huit bits, il faudrait donc un neuvième bit, et c'est l'indicateur de retenue qui va jouer ce rôle. Si C est à Ø au début d'une addition (C = Ø) et mis à 1 (C = 1) après l'opération, le résultat est supérieur à 255. Il s'ensuit que s'il est mis à 1 (C = 1) avant une soustraction, et qu'il vaut Ø après l'opération (C = Ø), cela veut dire que le nombre soustrait était plus grand que le premier nombre. Il existe deux instructions pour modifier l'indicateur de retenue :

CLC	Mettre l'indicateur de retenue à Ø (C = Ø)
SEC	Mettre l'indicateur de retenue à 1 (C = 1)

Il y a deux instructions qui utilisent l'indicateur de retenue :

BCC	Brancher s'il n'y a pas de retenue (si C = Ø)
BCS	Brancher s'il y a une retenue (si C = 1)

Chapitre 7

Les modes d'adressage I

Le 65Ø2 a un jeu d'instructions plutôt restreint, si on le compare à d'autres microprocesseurs. En fait, il dispose de 56 instructions de base. Mais un grand nombre d'entre elles peuvent être utilisées de plusieurs façons, ce qui porte en fait le nombre d'opérations possibles à 152. La manière dont ces instructions seront interprétées est déterminée par le *mode d'adressage* utilisé, dont voici quelques exemples.

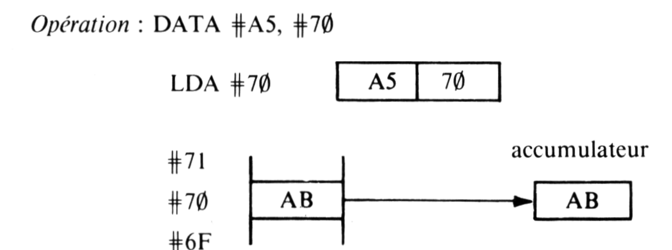
Mode d'adressage	Exemple de mnémonique	Code opérateur	Opérande(s)
Immédiat	LDA @#FF	A9	FF
En page zéro	LDA #7Ø	A5	7Ø
Indexé en page zéro	LDA #7Ø, X	B5	7Ø
Absolu	LDA #15ØØ	AD	ØØ 15
Pré-indexé indirect	LDA (#7Ø, X)	A1	7Ø
Post-indexé indirect	LDA (#7Ø), X	B1	7Ø
Indexé absolu	LDA #15ØØ, X	BD	ØØ 15

Ces sept instructions ont toutes pour effet de charger l'accumulateur, mais dans chaque cas la donnée chargée provient d'une source différente qui est définie par le code opérateur. Ce dernier, comme vous l'avez remarqué, est différent dans chaque cas. Pour le moment, nous ne nous intéresserons qu'aux deux premiers de ces modes d'adressage, l'adressage immédiat et l'adressage en page zéro, que nous avons déjà utilisés plusieurs fois.

L'adressage en page zéro

L'adressage en page zéro est utilisé pour désigner une adresse dans les 256 premiers octets de la RAM, où les données à charger dans le registre spécifié doivent se trouver. Comme l'octet haut de l'adresse est toujours #ØØ, on l'omet, ce qui fait que l'instruction et l'adresse ne nécessitent que deux octets de mémoire.

Opération : DATA #A5, #7Ø



Dans cet exemple (LDA #7Ø), le contenu de l'adresse #7Ø (c'est-à-dire ici #AB) est chargé dans l'accumulateur.

L'utilisation de l'adressage en page zéro nécessite quelques précautions, parce que l'interpréteur BASIC se sert de cette zone de mémoire comme d'un bloc-notes pour ranger des adresses et effectuer des calculs. Il ne devrait pas y avoir de problèmes si votre programme est écrit exclusivement en code machine. Mais il faut être particulièrement vigilant si vous utilisez à la fois BASIC et du code machine.

En effet, il peut arriver que votre code entre en conflit avec des données rangées en page zéro par l'interpréteur (les compteurs de boucles FOR... NEXT par exemple). Malheureusement, Oric ne fournit pas de documentation sur cette zone-clé de la mémoire. Par conséquent servez-vous autant que possible de la zone réservée à l'utilisateur (entre #4ØØ et #42Ø) pour ranger vos données. Si vous ne pouvez éviter la page zéro, testez soigneusement votre code machine, pour être sûr qu'il n'interfère pas avec BASIC. (Si c'était le cas, il faudrait le recharger, avec l'instruction CLOAD, et le tester avec des adresses différentes.

Le tableau 7.1 montre les instructions utilisant l'adressage en page zéro.

Tableau 7.1

Les instructions de l'adressage en page zéro

ADC	Addition avec retenue	LDA	Chargement de l'accumulateur
AND	ET logique	LDX	Chargement du registre X
ASL	Décalage arithmétique à gauche	LDY	Chargement du registre Y
BIT	Test de bit	LSR	Décalage logique à droite
CMP	Comparaison avec l'accumulateur	ORA	OU logique
		ROL	Rotation à gauche
CPX	Comparaison avec le registre X	ROR	Rotation à droite
CPY	Comparaison avec le registre Y	SBC	Soustraction avec retenue
DEC	Décrémentation de la mémoire	STA	Stockage de l'accumalateur
EOR	OU EXCLUSIF logique	STX	Stockage du registre X
INC	Incrémentation de la mémoire	STY	Stockage du registre Y

L'adressage immédiat

Ce mode d'adressage est utilisé lorsqu'on veut charger l'accumulateur ou les registres d'index avec une valeur donnée, connue au moment où l'on écrit le programme. Le 65Ø2 sait, grâce au code opérateur, que l'octet suivant est une donnée, et non une adresse. Mais pour que *nous* nous en souvenions, et pour nous aider à écrire en assembleur, on peut faire précéder l'octet de donnée par un signe distinctif, le «@» (qui se trouve sur la même touche que le 2 sur le clavier de l'Oric). On ne peut toutefois charger qu'un seul octet, parce que la taille du registre est limitée à huit bits. Si vous voulez que votre programme en code machine charge l'accumulateur avec la valeur 255, vous pouvez inclure la ligne suivante :

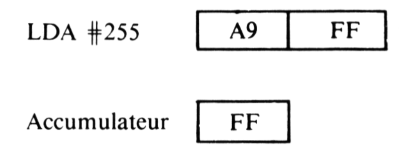
```
DATA #A9, #FF:REM LDA @#FF
```

où #A9 signifie «charger l'accumulateur de manière immédiate». On peut faire la même opération avec les registres X et Y :

```
DATA #A2, #41 :REM LDX @"A"
DATA #AØ, #FA:REM LDY @#FA
```

où A2 et AØ sont les codes d'adressage immédiat pour charger les registres X et Y.

Opération :



Le programme 3 utilise à la fois l'adressage en page zéro et l'adressage immédiat, pour afficher un point d'exclamation sur l'écran.

Programme 3

```
1Ø  REM * * Adressage immédiat et adressage en page zéro * *
2Ø  CODE = #4ØØ
3Ø  FOR INDEX = Ø TO 9
4Ø    READ OCTET
5Ø    POKE CODE + INDEX, OCTET
6Ø  NEXT INDEX
7Ø
8Ø  REM * * Les données du code machine * *
81  REM * Charge le code ASCII de "!" dans X *
82  REM * Range le code à l'adresse #22 *
83  REM * Charge le contenu de l'adresse #22 dans l'accumulateur *
84  REM * Range l'accumulateur dans la mémoire d'écran *
85  REM * Retour sous BASIC *
```

```
 9Ø  DATA #A2, #21          : REM LDX @"!"
1ØØ  DATA #86, #22          : REM STX #22
11Ø  DATA #A5, #22          : REM LDA #22
12Ø  DATA #8D, #AA, #BB     : REM STA ECRAN
13Ø  DATA #6Ø               : REM RTS
14Ø
15Ø  REM * * Initialisation et exécution * *
16Ø  CLS
17Ø  CALL (CODE)
```

Chapitre 8

Des bits et des octets

Charger, ranger, transférer

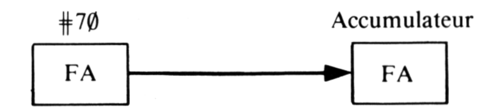
Pour pouvoir modifier et manipuler le contenu de la mémoire et des registres, nous disposons de trois ensembles d'instructions.

Les instructions de chargement

On appelle «charger» (LOAD en anglais) le fait de placer dans un registre une donnée contenue dans la mémoire. Nous avons déjà rencontré des exemples de cette opération. A titre de rappel, voici les trois instructions de chargement :

LDA	Charger l'accumulateur
LDX	Charger le registre X
LDY	Charger le registre Y

Ces instructions peuvent être employées toutes les trois avec l'adressage immédiat, mais en ce qui concerne la mémoire, il est plus correct de dire que le contenu d'une adresse est *copié* dans un registre, puisque l'emplacement d'origine n'est absolument pas modifié par cette opération. Par exemple LDA #7Ø copie le contenu de l'emplacement #7Ø (ici FA) dans l'accumulateur, sans modifier cet emplacement :



Les indicateurs Négatif et Zéro du registre d'état sont affectés par l'opération de chargement.

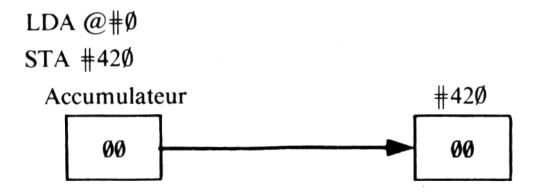
Les instructions de rangement

L'opération inverse, c'est-à-dire placer le contenu d'un registre en mémoire, s'appelle *ranger* (STORE en anglais). Il existe trois instructions de rangement :

STA Ranger le contenu de l'accumulateur
STX Ranger le contenu du registre X
STY Ranger le contenu du registre Y

La valeur du registre reste la même et aucun indicateur n'est affecté.

Exemple :



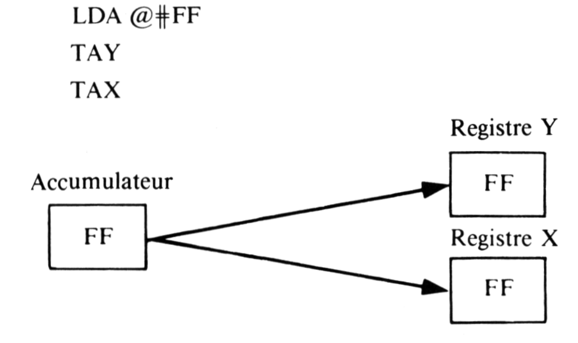
Les instructions de transfert

Il existe également des instructions qui permettent de copier le contenu d'un registre dans un autre : c'est ce qu'on appelle un *transfert*. Cette opération affecte les indicateurs Négatif et Zéro en fonction de la donnée transférée. Les instructions de transfert entre les registres d'index et l'accumulateur sont au nombre de quatre :

TXA Transférer le contenu du registre X dans l'accumulateur
TAX Transférer le contenu de l'accumulateur dans le registre X
TYA Transférer le contenu du registre Y dans l'accumulateur
TAY Transférer le contenu de l'accumulateur dans le registre Y

Exemple :

LDA @#FF
TAY
TAX



Malheureusement, il est impossible de transférer une donnée directement d'un registre d'index à un autre. Il faut passer par l'accumulateur.

```
YVERSX   TYA        \ Y dans l'accumulateur
         TAX        \ accumulateur dans X
```

de même :

```
XVERSY   TXA        \ X dans l'accumulateur
         TAY        \ accumulateur dans Y
```

Ce genre d'opération portant sur un seul octet est désigné par le terme d'*adressage inhérent*, puisque l'instruction contient toutes les informations nécessaires à son exécution.

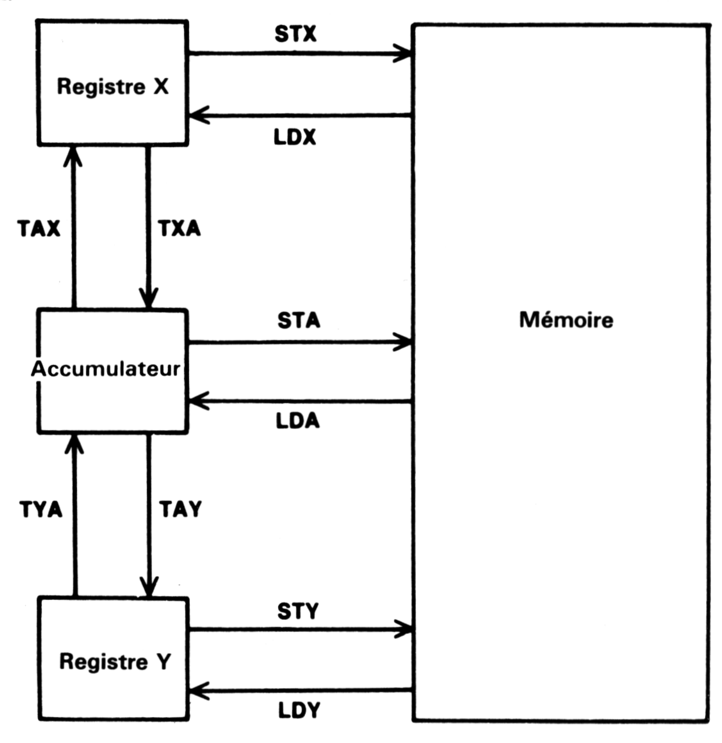


Figure 8.1 La circulation des données avec les instructions de chargement, de rangement et de transfert.

Les pages mémoire

Nous avons vu que le Compteur de programme se compose de deux registres de huit bits, ce qui fait un total de seize bits. Si tous ces bits sont à 1 (11111111 11111111) la valeur obtenue est 65535 ou #FFFF. Les adresses disponibles sur le 65Ø2 se situent donc entre #ØØØØ et #FFFF. Elles sont regroupées par *pages*, et le numéro de page est donné par le contenu de PCH. Par conséquent, c'est PCL qui indiquera l'emplacement à l'intérieur de la page.

Comme le montre la figure 8.2, chaque page de la mémoire peut être comparée à une page d'un livre. Ce livre comporte 256 pages numérotées en hexadécimal de #ØØ à #FF. Chaque page contient 256 lignes qui sont elles aussi numérotées (de haut en bas) de #ØØ à #FF.

Ainsi l'adresse #FFFF renvoie à la ligne #FF de la page #FF, c'est-à-dire la dernière adresse disponible dans la carte mémoire de l'Oric. A la différence d'un livre classique, la mémoire commence à la page #ØØ, plus connue sous le nom de *page zéro*. Etre situé en page zéro du 65Ø2 est très important, comme nous le verrons lorsque nous reviendrons sur les différents modes d'adressage au chapitre 10.

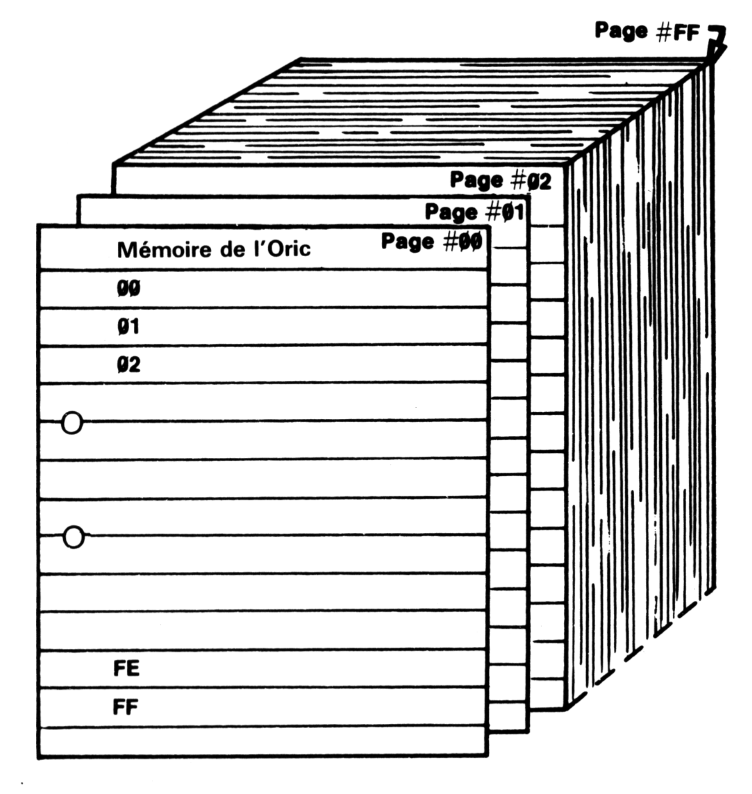


Figure 8.2 Les pages de la mémoire de l'Oric.

Bien que nous ayons comparé la carte mémoire de l'Oric à une suite de pages, on en parle plus souvent en terme de «K». Les Oric, par exemple, sont pourvus de 16K ou 48K de mémoire. La lettre «K» est l'abréviation de «kilo», mais à la différence du système métrique, un «kilo de mémoire», ou kilo-octet, équivaut à 1Ø24 octets, et non à 1ØØØ. Cette valeur, légèrement plus élevée, a été choisie parce qu'elle est divisible par 256 et correspond exactement à quatre pages en mémoire : (4 × 256 = 1Ø24). La mémoire totale sera donc de 64K puisque 65536/1Ø24 = 64. C'est d'ailleurs ce que vous confirmera le petit programme ci-dessous (écrit en Basic!).

Programme 4

```
1Ø  REM * * Calcul du nombre d'octets * *
2Ø  CLS
3Ø  REPEAT
4Ø  INPUT "Valeur de K :"; K
5Ø  OCTETS = 1Ø24 * K
6Ø  PRINT "K vaut"; OCTETS; "en décimal"
7Ø  PRINT HEX$(OCTETS); "en hexadécimal"
8Ø  UNTIL Ø
```

Chapitre 9

Un peu d'arithmétique en code machine

Nous pouvons maintenant appliquer quelques-uns des principes de base que nous avons vus dans les premiers chapitres à des fins un peu plus utiles : l'addition et la soustraction de nombres. Ces deux opérations sont fondamentales en code machine et interviennent dans la plupart des programmes.

L'addition

Deux instructions vont permettre d'effectuer des additions :

CLC Mise à Ø de l'indicateur de Retenue
ADC Addition avec retenue

La première de ces instructions, CLC, met simplement l'indicateur de Retenue à Ø, (C = Ø). En général, on commence toute addition par cette instruction, puisque ADC, qui va réellement effectuer l'addition, fait la somme de l'accumulateur, de l'octet spécifié et de l'indicateur de Retenue. Tout cela paraîtra plus clair, lorsque vous aurez vu quelques programmes d'addition simple. Entrez donc le programme 5.

Programme 5

```
1Ø  REM * * Addition simple * *
2Ø  CODE = #4ØØ
3Ø  FOR INDEX = Ø TO 7
4Ø    READ OCTET
5Ø    POKE CODE + INDEX, OCTET
6Ø  NEXT INDEX
```

```
70
80  REM * * Les données du code machine * *
90
100 DATA #18              : REM CLC
110 DATA #A9, #07         : REM LDA @#7
120 DATA #69, #03         : REM ADC @#3
130 DATA #85, #70         : REM STA #70
140 DATA #60              : REM RTS
150
160 CALL (CODE)
170 PRINT "La réponse est : ";
180 PRINT PEEK (#70)
```

Comme vous pouvez le voir en ligne 110 ce programme charge 7 dans l'accumulateur (adressage immédiat). L'adressage immédiat est réutilisé en ligne 120 pour ajouter 3 à la valeur de l'accumulateur. Puis le résultat, contenu dans l'accumulateur, est rangé à l'adresse #70. La ligne 160 exécute le code machine, et le résultat est affiché (si vous comptez très vite sur vos doigts, vous savez déjà qu'il était égal à 10). Faites tourner ce programme, puis essayez avec d'autres valeurs, en modifiant les lignes 110 et 120. Modifiez ensuite la ligne 100 ainsi :

```
100 DATA #38              : REM SEC
```

Cette instruction met l'indicateur de Retenue à un (C = 1) lors de la prochaine exécution du programme. Rétablissez les valeurs d'origine en ligne 110 et 120 (si vous les avez modifiées), et vous obtiendrez la valeur 11 comme résultat. Car la valeur de l'indicateur de retenue est prise en compte par l'instruction ADC (addition avec retenue), et cette valeur est à présent égale à 1.

	Accumulateur	+	mémoire	+	retenue	=	Résultat.
CLC	7	+	3	+	0	=	10
SEC	7	+	3	+	1	=	11

Vous pouvez essayer à nouveau avec des valeurs différentes, vous verrez que le résultat est toujours supérieur de 1 à celui que vous attendiez. Mais ce programme a deux inconvénients : il occupe beaucoup de place en mémoire et est relativement lent à l'exécution. Si l'on connaît au départ les valeurs à additionner, il est plus efficace de faire l'opération au préalable. La partie du programme en code machine peut à ce moment-là être réduite à deux lignes :

```
DATA #A9, #0A             : REM LDA @10
DATA #85, #22             : REM STA #22
```

Le programme 6 est un programme-type d'addition de nombres sur un seul octet.

Programme 6

```
1Ø  REM * * Addition de nombres sur un seul octet * *
2Ø  CODE = #4ØØ
3Ø  FOR INDEX = Ø TO 7
4Ø    READ OCTET
5Ø    POKE CODE + INDEX, OCTET
6Ø  NEXT INDEX
7Ø
8Ø  REM * *Les données du code machine * *
9Ø
1ØØ DATA #18           : REM CLC
11Ø DATA #A5, #7Ø      : REM LDA #7Ø
12Ø DATA #65, #71      : REM ADC #71
13Ø DATA #85, #72      : REM STA #72
14Ø DATA #6Ø           : REM RTS
15Ø
16Ø REM * * Initialisation et exécution * *
17Ø CLS 18Ø  INPUT "Premier nombre";NB1
19Ø POKE #7Ø, NB1
2ØØ INPUT "Deuxième nombre";NB2
21Ø POKE #71, NB2
22Ø CALL (CODE)
23Ø PRINT "La réponse est :";
24Ø PRINT PEEK (#72)
```

Exécutez ce programme plusieurs fois en donnant des petites valeurs numériques lorsqu'il vous les demande. Puis essayez en donnant 128 et 128. Le résultat est Ø parce que la réponse, 256, est trop grande pour être contenue dans un seul octet :

```
  128        #8Ø          1 Ø Ø Ø Ø Ø Ø Ø
+ 128      + #8Ø      +   1 Ø Ø Ø Ø Ø Ø Ø
-----      -----      -------------------
  256       #1ØØ      (1) Ø Ø Ø Ø Ø Ø Ø Ø
```

Comme on le constate, l'addition des deux bits les plus significatifs a produit une retenue, et comme l'indicateur de Retenue est mis à Ø au départ, il sera maintenant à 1 (C = 1), pour signaler que le résultat est trop grand pour tenir sur un octet. Ce principe est utilisé pour additionner des nombres de plus d'un octet et est illustré par le programme 7 qui additionne des nombres sur deux octets.

Programme 7

```
10  REM * * Addition de nombres sur deux octets * *
20  CODE = #400
30  FOR INDEX = 0 TO 13
40    READ OCTET
50    POKE CODE + INDEX, OCTET
60  NEXT INDEX
70
80  REM * * Les données du code machine * *
90
100 DATA #18           : REM CLC
110 DATA #A5, #70      : REM LDA #70
114 DATA #65, #72      : REM ADC #72
115 DATA #85, #74      : REM STA #74
121 DATA #A5, #71      : REM LDA #71
122 DATA #65, #73      : REM ADC #73
130 DATA #85, #75      : REM STA #75
140 DATA #60           : REM RTS
150
160 REM * * Initialisation et exécution * *
170 CLS
180 INPUT "Premier nombre"; NB1
190 DOKE #70, NB1
200 INPUT "Deuxième nombre"; NB2
210 DOKE #72, NB2
220 CALL (CODE)
230 PRINT "La réponse est :";
240 PRINT DEEK (#74)
```

Ce programme donnera un résultat correct pour tous les nombres dont la somme est inférieure à 65535 (#FFFF) qui est la plus grande valeur numérique qui puisse être contenue dans deux octets. Notez que l'indicateur de Retenue est mis à Ø dès le début du code machine. S'il se produit une retenue dans l'addition des deux octets bas, elle sera reportée sur l'addition des deux octets hauts.

La soustraction

Les deux instructions qui sont associées à la soustraction sont :

SEC   Mise à 1 de l'indicateur de Retenue
SBC   Soustraction avec retenue

L'opération qui consiste à retrancher un nombre d'un autre (ou à trouver leur différence), est l'inverse de celle qui consiste à additionner deux nombres, qui a été analysée dans les exemples précédents. Premièrement, l'indicateur de Retenue est mis à 1 (C = 1) avec l'instruction SEC, puis la valeur donnée est retranchée du contenu de l'accumulateur avec l'instruction SBC. Le résultat de la soustraction est dans l'accumulateur. Le programme suivant effectue une soustraction sur un seul octet :

Programme 8

```
 1Ø  REM * * Soustraction simple * *
 2Ø  CODE = #4ØØ
 3Ø  FOR INDEX = Ø TO 7
 4Ø    READ OCTET
 5Ø    POKE CODE + INDEX, OCTET
 6Ø  NEXT INDEX
 7Ø
 8Ø  REM * * Les données du code machine * *
 9Ø
1ØØ  DATA #38           : REM SEC
11Ø  DATA #A5, #7Ø      : REM LDA #7Ø
12Ø  DATA #E5, #71      : REM SBC #71
13Ø  DATA #85, #72      : REM STA #72
14Ø  DATA #6Ø           : REM RTS
15Ø
16Ø  REM * * Initialisation et exécution * *
17Ø  CLS
18Ø  INPUT "Premier nombre :";NB1
19Ø  POKE #7Ø, NB1
2ØØ  INPUT "Nombre à soustraire :";NB2
21Ø  POKE #71, NB2
22Ø  CALL (CODE)
23Ø  PRINT "La réponse est :";
24Ø  PRINT PEEK (#72)
```

Faites tourner le programme en modifiant les valeurs, pour voir les résultats. Peut-être êtes-vous surpris de voir l'indicateur de Retenue mis à 1 avant une soustrac-

tion. Si vous vous reportez au chapitre 3, vous verrez qu'une soustraction se fait en additionnant la valeur du complément à deux et qu'on obtient le complément à deux en inversant tous les bits et en ajoutant 1. Le 65∅2 tire ce 1 de l'indicateur de Retenue. Ce qui nous permet de déduire que :

1. Si l'indicateur de Retenue est à 1 après SBC, le résultat est positif ou nul.
2. Si l'indicateur de Retenue est à ∅ après SBC, le résultat est négatif et l'on a eu besoin d'une retenue pour effectuer la soustraction.

Modifions la ligne 1∅∅ pour faire CLC et exécutons le programme. On obtient maintenant le résultat attendu, moins 1 : c'est tout simplement que le 65∅2 n'a jamais pu obtenir le complément à deux, puisqu'il n'a trouvé qu'un ∅ en additionnant le contenu de l'indicateur de Retenue à la valeur du complément à 1.

Pour soustraire des nombres de deux octets, on met l'indicateur de retenue à 1 au début de la routine, et on soustrait les octets correspondants en rangeant les résultats au fur et à mesure. Le programme qui effectue cette opération ressemble un peu à ceci :

Programme 9

```
 1∅  REM * * Soustraction de nombres sur deux octets * *
 2∅  CODE = #4∅∅
 3∅  FOR INDEX = ∅ TO 13
 4∅    READ OCTET
 5∅    POKE CODE + INDEX, OCTET
 6∅  NEXT INDEX
 7∅
 8∅  REM * * Données du code machine * *
 9∅
1∅∅  DATA #38            : REM SEC
11∅  DATA #A5, #7∅       : REM LDA #7∅
12∅  DATA #E5, #72       : REM SBC #72
13∅  DATA #85, #74       : REM STA #74
14∅  DATA #A5, #71       : REM LDA #71
15∅  DATA #E5, #73       : REM SBC #73
16∅  DATA #85, #75       : REM STA #75
17∅  DATA #6∅            : REM RTS
18∅
19∅  REM * * Initialisation et exécution * *
2∅∅  CLS
21∅  INPUT "Premier nombre :";NB1
22∅  DOKE #7∅, NB1
```

```
23Ø INPUT "Nombre à soustraire :";NB2
24Ø DOKE #72, NB2
25Ø CALL (CODE)
26Ø PRINT "Le résultat est :";
27Ø PRINT DEEK (#74)
```

Le complément à deux

L'instruction SBC peut être utilisée pour obtenir la valeur du complément à deux d'un nombre. On l'obtient en retranchant ce nombre de zéro. Le programme suivant vous permet d'obtenir, à partir d'un nombre décimal (inférieur à 255), la valeur de son complément à deux :

Programme 10

```
 1Ø REM * * Le complément à deux d'un nombre * *
 2Ø CODE = #4ØØ
 3Ø FOR INDEX = Ø TO 7
 4Ø   READ OCTET
 5Ø   POKE CODE + INDEX, OCTET
 6Ø NEXT INDEX
 7Ø
 8Ø REM * * Données du code machine * *
 9Ø
1ØØ DATA #38            : REM SEC
11Ø DATA #A9, #ØØ       : REM LDA @Ø
12Ø DATA #E5, #7Ø       : REM SBC #7Ø
13Ø DATA #85, #71       : REM STA #71
14Ø DATA #6Ø            : REM RTS
15Ø
16Ø REM * * Initialisation et exécution * *
17Ø
18Ø INPUT "Nombre à convertir :";NB
19Ø POKE #7Ø, NB
2ØØ CALL (CODE)
21Ø PRINT "La valeur du complément à deux est :";
22Ø PRINT PEEK (#71)
```

Chapitre 10

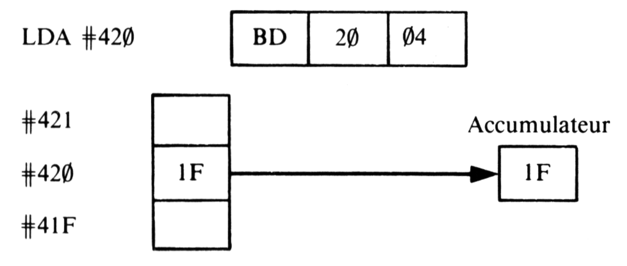
Les modes d'adressage II

Revenons un peu aux différents modes d'adressage. Dans les chapitres précédents, nous avons vu comment on pouvait obtenir directement une donnée, en utilisant *l'adressage immédiat*, ou indirectement, en utilisant l'adressage *en page zéro*. Nous allons voir maintenant comment il est possible d'accéder au reste de la mémoire en utilisant une adresse sur deux octets, à la fois directement et indirectement (en passant dans ce dernier cas par la page zéro), et comment on peut manipuler des blocs de mémoire en se servant de l'adressage *indexé*.

L'adressage absolu

L'adressage absolu fonctionne exactement comme l'adressage en page zéro, mais il renvoie à tous les autres emplacements de la mémoire (c'est-à-dire entre les adresses #1ØØ et #FFFF). Le mnémonique est suivi de l'adresse notée sur deux octets.

Opération :



Comme on peut le voir, le code opérateur est suivi de l'adresse, qui comme toujours, est rangée à l'envers, c'est-à-dire en commençant par l'octet de poids faible. Le contenu de l'adresse #42Ø est copié dans l'accumulateur au moment où cette instruction s'exécute. Le programme 3 (dans le chapitre 7) utilisait l'adressage absolu en ligne 12Ø pour ranger le contenu de l'accumulateur dans la mémoire de l'écran, provoquant ainsi l'affichage d'un point d'exclamation.

Le tableau 10.1 présente l'ensemble des instructions qui utilisent l'adressage absolu.

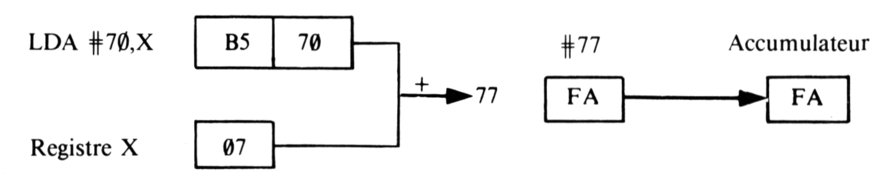
Tableau 10.1

Les instructions utilisant l'adressage absolu			
ADC	Addition avec retenue	LDA	Chargement de l'accumulateur
AND	ET logique	LDX	Chargement du registre X
ASL	Décalage arithmétique à gauche	LDY	Chargement du registre Y
BIT	Test de bit	LSR	Décalage logique à droite
CMP	Comparaison avec l'accumulateur	ORA	OU logique
CPX	Comparaison avec le registre X	ROL	Rotation à gauche
CPY	Comparaison avec le registre Y	ROR	Rotation à droite
DEC	Décrémentation de la mémoire	SBC	Soustraction avec retenue
EOR	OU EXCLUSIF logique	STA	Rangement de l'accumulateur
INC	Incrémentation de la mémoire	STX	Rangement du registre X
JMP	Branchement	STY	Rangement du registre Y
JSR	Appel sous-programme		

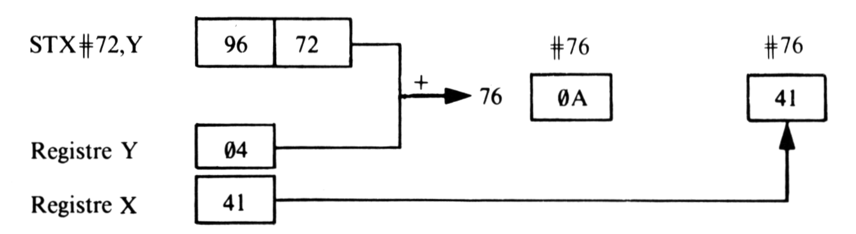
L'adressage indexé en page zéro

Dans l'adressage indexé en page zéro, l'adresse d'un opérande est obtenue en ajoutant le contenu de l'un des registres d'index (X ou Y) à l'adresse en page zéro spécifiée.

Opération :



Dans cet exemple le registre X contient #Ø7. On ajoute cette valeur à celle de l'adresse spécifiée, #7Ø, pour obtenir l'adresse effective, #77. Le contenu de l'adresse #77 (ici FA) est alors chargé dans l'accumulateur. De la même façon :



Ici le registre Y sert d'index pour ranger le contenu du registre X à l'adresse #76. On obtient cette adresse en additionnant la valeur du registre Y (#Ø4) à la valeur indiquée (#72). Le contenu précédent de l'adresse #76 (ØA) est détruit.

Les instructions d'adressage indexé en page zéro sont énumérées dans le tableau 10.2

Tableau 10.2

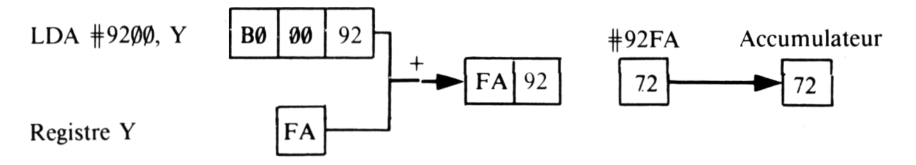
Les instructions utilisant l'adressage indexé en page zéro			
ADC	Addition avec retenue	LDY	Chargement du registre Y
AND	ET logique	LSR	Décalage logique à droite
ASL	Décalage arithmétique à gauche	ORA	OU logique
CMP	Comparaison avec l'accumulateur	ROL	Rotation à gauche
DEC	Décrémentation de la mémoire	ROR	Rotation à droite
EOR	OU EXCLUSIF logique	SBC	Soustraction avec retenue
INC	Incrémentation de la mémoire	STA	Rangement de l'accumulateur
LDA	Chargement de l'accumulateur	* STX	Rangement du registre X
* LDX	Chargement du registre X	STY	Rangement du registre Y

Le signe * marque les seules commandes qui peuvent utiliser le registre Y comme index. Toutes les autres ne peuvent utiliser que le registre X.

L'adressage indexé absolu

L'adressage indexé absolu ressemble à l'adressage indexé en page zéro, à la différence qu'il donne accès à tous les emplacements de la mémoire en dehors de la page zéro. Les registres X et Y peuvent être utilisés selon les cas, comme index pour l'accumulateur, ou l'un pour l'autre.

Opération :



Le contenu du registre Y (#FA) est ajouté à l'adresse sur deux octets (#92ØØ) pour produire l'adresse réelle (#92FA). Le programme suivant montre comment on peut utiliser l'adressage indexé absolu pour transférer une partie de la mémoire de l'écran d'un emplacement à un autre.

Programme 11

```
1Ø  REM * * Adressage indexé absolu * *
2Ø  REM * * Réinitialiser HIMEM #96ØØ * *
3Ø  CODE = #96ØØ
4Ø  FOR INDEX = Ø TO 16
```

```
50   READ OCTET
60   POKE CODE + INDEX, OCTET
70  NEXT INDEX
80
90  REM * * Les données du code machine * *
100
110  DATA #A2, #00           : REM LDX @0
120  DATA #A0, #17           : REM LDY @#17
130  DATA #BD, #AA, #BB      : REM LDA #BBAA, X
140  DATA #09, # 20          : REM ORA @#20
150  DATA #9D, #EA, #BC      : REM STA #BCEA, X
160  DATA #E8                : REM INX
170  DATA #88                : REM DEY
180  DATA #D0, #F4           : REM BNE - 12
190  DATA #60                : REM RTS
200
210  REM * * Initialisation et exécution * *
220  CLS
230  PRINT "ADRESSAGE INDEXE ABSOLU"
240  GET A$ : REM Actionnez une touche
250  CALL (CODE)
```

A l'exécution, le message de la ligne 230 s'affiche à l'écran en mode TEXT, puis le programme attend que l'on appuie sur une touche pour appeler le code machine. Le registre X fait office de compteur de déplacement et le registre Y sert comme compteur de boucle. Ils sont initialisés en lignes 110 et 120. L'octet situé à l'adresse «haut de l'écran + X» est chargé dans l'accumulateur, en utilisant l'adressage indexé absolu (ligne 130). On effectue ensuite un OR entre cet octet, qui contient en fait le code ASCII du caractère que l'on veut déplacer et la valeur #20. Cette opération a pour effet de transformer ce caractère en minuscule, en forçant le bit 5 à 1 (ce qui fait la différence entre la représentation binaire du code ASCII d'un caractère majuscule et celui du même caractère minuscule). Par exemple :

```
ASC "A" = #41   0 1 0 0 0 0 0 1
ASC "a" = #61   0 1 1 0 0 0 0 1
```

En ligne 150 on range cette nouvelle valeur dans la moitié inférieure de l'écran en utilisant à nouveau l'adressage indexé absolu. Le registre X est incrémenté (en ligne 160) pour pointer sur l'octet suivant; le registre Y est décrémenté (en ligne 170), et si son contenu est différent de zéro, le programme retourne en ligne 130 pour chercher l'octet suivant. Lorsque le registre Y atteint la valeur 0, le code machine redonne le contrôle à BASIC (ligne 190).

Les instruction liées à l'adressage indexé absolu sont décrites dans le tableau 10.3.

Tableau 10.3

Les instructions utilisant l'adressage indexé absolu			
* ADC	Addition avec retenue	* * LDX	Chargement du registre X
* AND	ET logique	LDY	Chargement du registre Y
ASL	Décalage arithmétique à gauche	LSR	Décalage logique à droite
* CMP	Comparaison avec l'accumulateur	* ORA	OU logique
DEC	Décrémentation de la mémoire	ROL	Rotation à gauche
* EOR	OU EXCLUSIF logique	ROR	Rotation à droite
INC	Incrémentation de la mémoire	* SBC	Soustraction avec retenue
* LDA	Chargement de l'accumulateur	* STA	Rangement de l'accumulateur

Les commandes sans * ne sont utilisables qu'avec le registre X. Les commandes marquées d'une * peuvent utiliser les deux registres, et celle marquée de deux * * ne peut utiliser que le registre Y.

L'adressage indirect

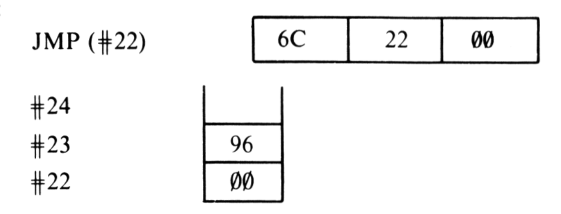
L'adressage indirect nous permet d'écrire ou de lire en mémoire à une adresse que l'on ignore au moment où on écrit le programme! Etonnant, non? En fait, le programme peut calculer lui-même l'adresse dont il a besoin. Autrement dit, un programme peut contenir plusieurs tables de données qui seront toutes traitées de la même manière. Plutôt que d'écrire une routine pour chaque table, on peut développer une routine-type, à laquelle on fournira l'adresse de la table à traiter avant chaque exécution.

Ce qui rend l'adressage indirect si séduisant, c'est qu'il permet d'accéder à toute la carte de la mémoire de l'Oric avec une seule instruction de deux octets. Pour distinguer l'adressage indirect des autres modes d'adressage, les opérandes devront être écrits entre parenthèses. On ne peut utiliser l'adressage indirect au sens strict que pour une seule instruction, dont le mnémonique est JMP. Nous verrons les fonctions de cette instruction en détail au chapitre 13. Pour le moment, bornons-nous à dire qu'elle est, pour le 65Ø2, l'équivalent de GOTO en BASIC. (Bien sûr, elle envoie à une adresse et non à une ligne de programme). Une instruction de branchement (JuMP en anglais) aura souvent la forme :

JMP (#22)

L'adresse spécifiée dans l'instruction est celle de l'emplacement où est rangée l'adresse à laquelle le branchement doit se faire, et non directement celle-ci. En d'autres termes, une telle instruction signifie «ne vous branchez pas sur l'adresse spécifiée ici, mais allez-y pour trouver l'adresse de branchement».

Opération :



Dans cet exemple, nous voyons que #22 est l'adresse de l'octet bas d'une autre adresse dont l'octet haut est en #23. Ces deux emplacements servent temporairement à stocker cette deuxième adresse : on les appelle un *vecteur*. En exécutant JMP(#22) dans ce cas, on transfèrera l'exécution du programme à l'adresse #96ØØ. Le programme 12 est un exemple d'utilisation de cette instruction. Le code machine de ce programme est rangé dans deux zones différentes de la mémoire, identifiées par VECT et CODE (en lignes 2Ø et 3Ø). Le code machine rangé à partir de VECT gère l'initialisation du vecteur. Il utilise pour cela deux octets en page zéro (#7Ø et #71), et effectue l'équivalent d'un DOKE pour écrire la valeur #96ØØ à ces deux adresses. Notez que, comme toujours avec le 65Ø2, l'octet bas est rangé en premier. C'est en ligne 18Ø que s'effectue le branchement indirect à l'adresse #96ØØ par l'intermédiaire de #7Ø. Le code machine rangé en #96ØØ affiche alors une rangée d'astérisques sur la première ligne de l'écran en mode TEXT.

Programme 12

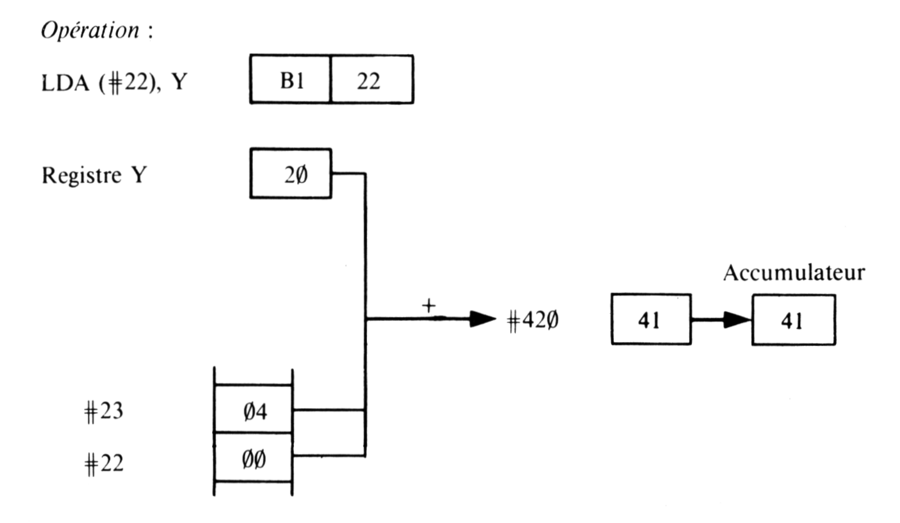
```
 1Ø REM * * Branchement indirect * *
 2Ø VECT = #4ØØ
 3Ø CODE = #96ØØ
 4Ø FOR INDEX = Ø TO 1Ø
 5Ø   READ OCTET
 6Ø   POKE VECT + INDEX, OCTET
 7Ø NEXT INDEX
 8Ø FOR INDEX = Ø TO 1Ø
 9Ø   READ OCTET
1ØØ   POKE CODE + INDEX, OCTET
11Ø NEXT INDEX
12Ø
13Ø REM * * Les données du code machine * *
135 REM * * Les données pour VECT * *
14Ø DATA #A9, #ØØ          : REM LDA @Ø
15Ø DATA #85, #7Ø          : REM STA #7Ø
16Ø DATA #A9, #96          : REM LDA @#96
17Ø DATA #85, #71          : REM STA #71
18Ø DATA #6C, #7Ø, #ØØ     : REM JMP (#7Ø)
```

```
190  REM * * Les données pour CODE * *
200  DATA #A2, #28        : REM LDX @#28
210  DATA #A9, #2A        : REM LDA @"*"
220  DATA #9D, #A7, #BB : REM STA #BBA7, X
230  DATA #CA             : REM DEX
240  DATA #D0, #FA        : REM BNE – 6
250  DATA #60             : REM RTS
260
270  REM * * Initialisation et exécution * *
280  CLS
290  CALL (VECT)
```

L'adressage post-indexé indirect

L'adressage post-indexé ressemble un peu à l'adressage indexé absolu, mais ici l'adresse de base est rangée dans un vecteur en page zéro, auquel on accède indirectement.

Opération :



Dans cet exemple, l'adresse de base est rangée dans le vecteur, aux adresses #22 et #23. Le contenu du registre Y (#2Ø) est ajouté à la valeur du vecteur (#4ØØ) pour donner l'adresse réelle (#42Ø) de la donnée. Il devrait être clair que cette forme d'adressage indexé donne accès à des blocs de 256 octets. Dans l'exemple ci-dessus, toute adresse située entre #4ØØ et #4FF est accessible en chargeant le registre Y en conséquence. Le programme 13 utilise l'adressage post-indexé indirect pour déplacer une ligne de la mémoire de l'écran de la moitié supérieure vers la moitié inférieure de l'écran.

Programme 13

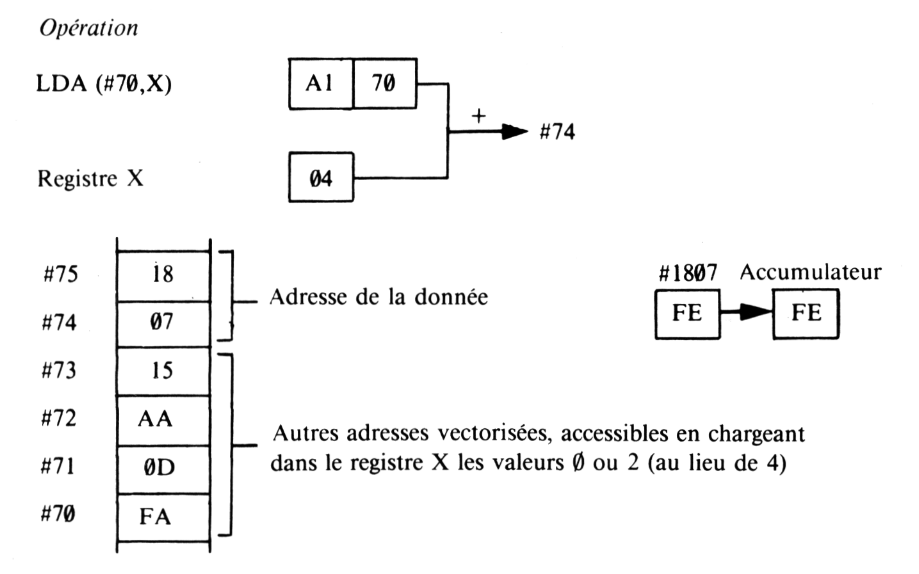
```
10  REM * * Adressage indirect * *
20  CODE = #400
30  FOR INDEX = 0 TO 12
40    READ OCTET
50    POKE CODE + INDEX, OCTET
60  NEXT INDEX
70
80  REM * * Les données du code machine * *
90
100 DATA #A0, #00          : REM LDY @#00
110 DATA #A2, #27          : REM LDX @#27
120 DATA #B1, #70          : REM LDA (#70), Y
130 DATA #91, #72          : REM STA (#72), Y
140 DATA #C8               : REM INY
150 DATA #CA               : REM DEX
160 DATA #D0, #F8          : REM BNE - 9
170 DATA #60               : REM RTS
180
190 REM * * Initialisation et exécution * *
200 CLS
210 DOKE #70, #BBAA        : REM haut de l'écran
220 DOKE #72, #BCEA        : REM bas de l'écran
230 PRINT "Adressage indirect"
240 CALL (CODE)
```

A l'exécution les deux adresses correspondant au haut et au bas de l'écran sont écrites dans quatre octets de la page zéro, à partir de l'adresse #70. Après avoir effacé l'écran, le programme affiche le titre en haut de l'écran (ligne 230).Le code machine commence par initialiser le registre Y avec la valeur 0 (ligne 100) et il charge ensuite dans le registre X le nombre d'octets à déplacer (ligne 110). En utilisant l'adressage indirect, on charge dans l'accumulateur l'octet qui se trouve à l'adresse obtenue en additionnant le contenu du registre Y au vecteur contenant l'adresse du haut de l'écran (ligne 120). Cet octet est ensuite rangée à l'adresse résultant de la somme du contenu du registre Y et du vecteur correspondant au bas de l'écran (ligne 130). Le registre Y est incrémenté en ligne 140 et le registre X est décrémenté en ligne 150. En ligne 160 on le teste pour savoir s'il est égal à zéro.

L'adressage pré-indexé indirect

Ce mode d'adressage est utilisé lorsqu'on veut accéder indirectement à un ensemble d'adresses absolues rangées en page zéro.

Opération



Ici, le contenu du registre X (#Ø4) est ajouté à l'adresse en page zéro (#7Ø) pour donner l'adresse du vecteur (#74). Les deux octets qu'il contient sont alors interprétés comme étant l'adresse de la donnée (#18Ø7). En chargeant le registre X avec la valeur #Ø2 on obtiendrait l'adresse #15AA. Le Tableau 10.4 donne la liste des instructions utilisant l'adressage pré et post-indexé.

Tableau 10.4

Les instructions utilisant les adressages pré et post-indexés indirects	
ADC	Addition avec retenue
AND	ET logique
CMP	Comparaison avec la mémoire
EOR	OU EXCLUSIF logique
LDA	Chargement de l'accumulateur
ORA	OU logique
SBC	Soustraction avec retenue
STA	Rangement de l'accumulateur

Les adressages inhérent et relatif

Il existe sur l'Oric deux autres modes d'adressage, à savoir l'*adressage inhérent* et l'*adressage relatif.* Nous nous en occuperons au cours des prochains chapitres.

Les tables

Les tables sont une méthode efficace, simple et rapide, pour obtenir des données qui exigeraient autrement un programme lourd et complexe en code machine (comme pour résoudre des équations). Supposons par exemple, que nous voulions développer un programme pour convertir des degrés Centigrades en degrés Fahrenheit. La formule est :

°F = 1.8 (°C) + 32

Comme vous pouvez le constater, cela nécessite une multiplication décimale (1.8 × C), puis une addition. Tout cela prend beaucoup de temps et met votre matière grise à rude épreuve! En plaçant dans une table une suite de valeurs de degrés Fahrenheit préalablement calculées, on pourra en extraire la valeur recherchée en se servant de la valeur en degrés Centigrades comme d'un index.

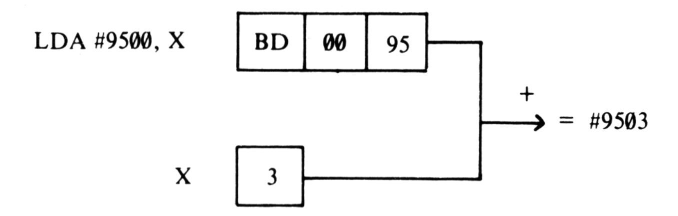
Programme 14

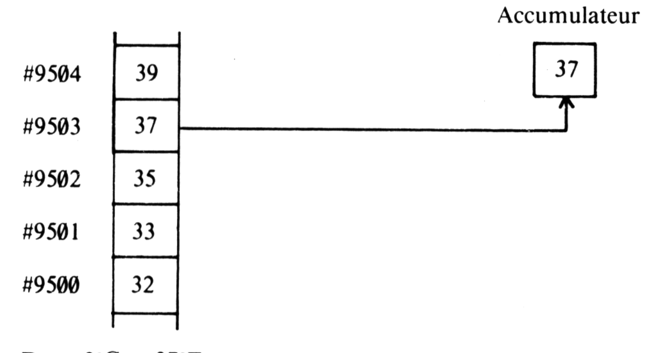
```
 1Ø  REM * * Conversion Centigrades en Fahrenheit * *
 2Ø  REM * * HIMEM #96ØØ * *
 3Ø  CODE = #96ØØ
 4Ø  FOR INDEX = Ø to 9
 5Ø    READ OCTET
 6Ø    POKE CODE + INDEX, OCTET
 7Ø  NEXT INDEX
 8Ø  REM * * Installation de la table de conversion * *
 85  TABLE = #95ØØ
 9Ø  FOR N = Ø TO 1ØØ
1ØØ    DEGRE = (1.8 * N) + 32
11Ø    POKE TABLE + N, DEGRE
12Ø  NEXT N
13Ø
14Ø  REM * * Les données du code machine * *
15Ø  DATA #AE, #ØØ, #Ø4  : REM LDX #4ØØ
16Ø  DATA #BD, #ØØ, #95  : REM LDA #95ØØ, X
17Ø  DATA #8D, #ØØ, #Ø4  : REM STA #4ØØ
18Ø  DATA #6Ø            : REM RTS
19Ø
2ØØ  REM * * Initialisation et exécution * *
2Ø5  CLS
21Ø  REPEAT
```

```
220     INPUT "Valeur en centigrades :";C
230     POKE #400, C
240     CALL (CODE)
250     PRINT "Valeur en Fahrenheit :";
260     PRINT PEEK (#400)
270   UNTIL C > 100
```

Les lignes 80 à 120 calculent les équivalents en degrés Fahrenheit des degrés Centigrades de 0 à 100, et les écrit en mémoire pour former une table qui commence à l'adresse #9500. L'exemple ci-dessous montre comment la valeur en Centigrades sert d'index pour trouver l'équivalent en Fahrenheit.

Exemple : Convertir 3°Centigrades en degrés Fahrenheit.





Donc 3°C = 37°F.

Chapitre 11

La pile

La pile est peut-être l'un des aspects du 65Ø2 les plus difficiles à maîtriser. Mais le temps que vous passerez à assimiler ce chapitre ne sera pas perdu, puisque vos programmes y gagneront en efficacité. La pile utilise toute la *Page #Ø1* (c'est-à-dire à tous les octets d'adresse #1ØØ à #1FF).

Elle sert à stocker temporairement des données ou des adresses, dont le programme aura à se resservir par la suite. Dans la plupart des cas, son rôle n'apparaît pas explicitement dans un programme. Par exemple, lorsqu'un programme BASIC fait appel à une instruction de type GOSUB, l'adresse de l'instruction BASIC suivante est rangée sur la pile, pour que l'exécution puisse se poursuivre à la sortie du sous-programme. Quand on place une donnée sur la pile, on dit qu'on l'*empile*, et quand on l'en extrait, on dit qu'on la *dépile*.

Il est important de comprendre que la pile a une structure de type LIFO (Last-In First-Out, dernier entré premier sorti), c'est à dire que le dernier octet empilé est obligatoirement le premier à être dépilé. On peut comparer la pile à un train garé contre un butoir. Supposez qu'on y raccroche les wagons 1, 2 et 3. (Voir figure 11.1). Le wagon n°1 est celui qui se trouve le plus près du butoir, le second est en milieu de train et le n°3 est en queue.

Le wagon n°3 est entré le dernier, il doit donc ressortir le premier.

3 2 1

Figure 11.1 Structure LIFO de la pile

Il est évident que le premier wagon qu'il sera possible de déplacer sera celui qu'on a garé en dernier lieu ; on pourra ensuite déplacer le wagon n°2, et enfin dégager le wagon n°1. Le *Pointeur de Pile* est un registre du 65Ø2 qui indique la première place libre sur la pile. Comme l'emplacement de la pile est fixé par le microprocesseur, on peut omettre la référence de la «page mémoire» (#Ø1), et le Pointeur de Pile indique automatiquement le prochain emplacement libre sur la pile. Lorsqu'on met l'Oric sous tension, ou après un BREAK, le Pointeur de Pile est chargé avec la valeur #FF, il indique le sommet de la pile. Cela signifie que la pile se remplit de haut en bas, et non de bas en haut comme on aurait pu le penser. Chaque fois que l'on empile une donnée, le Pointeur de Pile est décrémenté, et à l'inverse, il est incrémenté à chaque dépilement.

Les instructions de sauvegarde de données sur la pile

Le 65Ø2 dispose de quatre instructions pour empiler et dépiler l'accumulateur et le registre d'Etat :

PHA Empiler (PusH en anglais) l'Accumulateur
PLA Dépiler (PulL en anglais) dans l'Accumulateur
PHP Empiler le registre d'Etat
PLP Dépiler dans le registre d'Etat

Ces quatre instructions utilisent l'adressage inhérent et sont codées sur un seul octet.

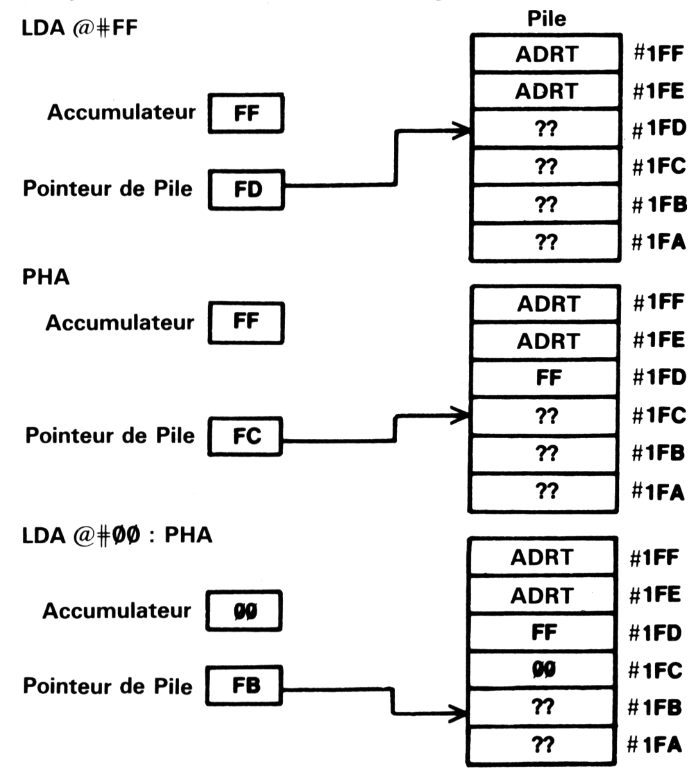


Figure 11.2 L'Empilement

Les instructions PHA et PHP fonctionnent de manière comparable, mais avec des registres différents. Dans les deux cas le registre d'origine n'est pas modifié. De même, PLA et PLP sont des opérations similaires, mais PLA n'affecte que les indicateurs Négatif et Zéro, alors que PLP affecte tous les indicateurs du registre d'état. Considérons cette suite d'instructions :

```
DEUXEMP   LDA @#FF          \ Charge #FF dans l'accumulateur
          PHA               \ Empile #FF
          LDA @#ØØ          \ Charge #ØØ dans l'accumulateur
          PHA               \ Empile #ØØ
```

La figure 11.2 montre exactement ce qui se produit lorsque ce programme est exécuté. Au début, le Pointeur de Pile contient #FD et indique l'emplacement libre suivant dans la pile. Les deux premiers emplacements #FF et #FE contiennent les octets de l'adresse de retour (ADRT) à laquelle le code machine passera éventuellement le contrôle. (Ce sera, par exemple, l'adresse de l'instruction suivante en BASIC si un programme en BASIC a fait appel à un sous-programme en code machine.) Les autres emplacements de la pile sont disponibles, ils sont représentés par ??.

Après avoir chargé #FF dans l'accumulateur, PHA le copie sur la pile. Notez que cette opération ne modifie pas le contenu de l'accumulateur. Après cette opération, le Pointeur de Pile est décrémenté pour indiquer l'emplacement libre suivant (#FC). Puis on charge #ØØ dans l'accumulateur et on l'empile à l'emplacement #FC. A nouveau, le Pointeur de Pile, après décrémentation, indique l'emplacement suivant (#FB). Pour dépiler ces données, on peut procéder ainsi :

```
DEUXDEP   PLA       \ Dépiler #ØØ
          STA #7Ø   \ Sauvegarder le contenu de l'accumulateur
          PLA       \ Dépiler #FF
          STA #71   \ Sauvegarder le contenu de l'accumulateur
```

La figure 11.3 illustre ce qui se passe dans ce cas. Le premier PLA met dans l'accumulateur la donnée qui a été empilée en dernier lieu, c'est-à-dire ici #ØØ. Le Pointeur de pile est incrémenté et indique le nouvel emplacement libre, #FC. Comme vous pouvez le voir sur le diagramme, le contenu de la pile n'est pas modifié, mais #ØØ sera détruit à la prochaine instruction d'empilement. STA #7Ø sauvegarde la valeur chargée dans l'accumulateur en mémoire, pour éviter qu'elle soit détruite par le prochain PLA. Ce deuxième PLA charge #FF dans l'accumulateur et incrémente à nouveau le Pointeur de Pile.

A l'évidence, il est fondamental de se souvenir de l'ordre dans lequel les données ont été empilées, puisqu'on devra *obligatoirement* les dépiler dans l'ordre inverse. Si vous ne respectez pas cette contrainte, vous risquez de provoquer des erreurs, ou même de bloquer votre programme. Le programme suivant montre comment se servir de la pile pour sauvegarder différents registres et afficher leur contenu à un autre moment. C'est particulièrement utile pour mettre au point les programmes qui, bizarrement, refusent de tourner.

```
SAUVREG   PHP   \ Empile le contenu du registre d'Etat
          PHA   \ Empile le contenu de l'accumulateur
          TXA   \ Transfère le contenu du registre X dans l'accumulateur
          PHA   \ Empile le contenu de l'accumulateur (X)
          TYA   \ Transfère le contenu du registre Y dans l'accumulateur
          PHA   \ Empile le contenu de l'accumulateur (Y)
```

Il est très important de sauvegarder le contenu des différents registres dans cet ordre. Le registre d'Etat doit être empilé en premier pour éviter une modification de son contenu due aux transferts suivants qui affectent les indicateurs Négatif et Zéro. De même, il faut sauvegarder le contenu de l'accumulateur pour ne pas le détruire en y transférant l'un des registres d'index.

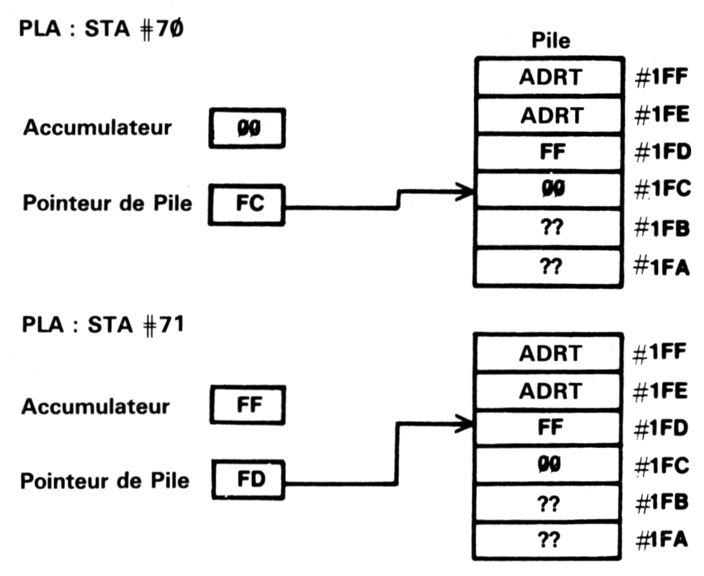


Figure 11.3 Le Dépilement

Si on sauvegarde ainsi les valeurs des différents registres, avant l'exécution d'une autre partie du programme, on pourra les récupérer en faisant :

```
PLA     \ Dépile dans l'accumulateur (Y)
TAY     \ Transfère A dans le registre Y
PLA     \ Dépile dans l'accumulateur (X)
TAX     \ Transfère A dans le registre X
PLA     \ Dépile dans l'accumulateur
PLP     \ Dépile dans le registre d'Etat
```

Il existe en plus deux autres instructions liées à la pile :

TSX   Transfère le Pointeur de Pile dans le registre X
TXS   Transfère le registre X dans le Pointeur de Pile

Nous verrons comment et pourquoi sauvegarder des adresses dans la pile au cours du chapitre 13.

Chapitre 12

Les boucles

Une boucle permet de répéter une partie d'un programme autant de fois qu'on le veut. Par exemple, sous BASIC, on peut faire afficher dix fois un « ! » en utilisant une boucle FOR...NEXT :

```
10  FOR NOMBRE = 0 TO 9
20  PRINT "!";
30  NEXT NOMBRE
```

En ligne 10, un compteur, appelé NOMBRE, est déclaré et initialisé à 0. En ligne 20 on affiche un « ! » et en ligne 30 on vérifie si NOMBRE a atteint la limite maximale qui lui a été assignée. Si ce n'est pas la cas, le programme incrémente la variable NOMBRE et affiche un autre point d'exclamation. Pour écrire ce genre de boucle en assembleur, il faut savoir comment contrôler les trois types d'opération décrits, c'est-à-dire les compteurs, les comparaisons et les branchements.

Les compteurs

Habituellement on se sert des registres d'index comme compteurs, parce qu'ils ont leurs propres instructions d'incrémentation et de décrémentation.

INX	Incrémente le registre X	X = X + 1
INY	Incrémente le registre Y	Y = Y + 1
DEX	Décrémente le registre X	X = X − 1
DEY	Décrémente le registre Y	Y = Y − 1

Toutes ces instructions affectent les indicateurs Négatif et Zéro. L'indicateur Négatif est mis à un si le bit le plus significatif du registre est à un après une incrémentation ou une décrémentation ; dans les autres cas, il est mis à 0. L'indicateur Zéro sera à 1 seulement si le registre concerné contient 0 après l'une de ces instructions.

Notez qu'incrémenter un registre qui contient #FF a pour effet de lui donner la valeur #00, de mettre l'indicateur Négatif à 0 (N = 0), et de positionner l'indica-

teur Zéro (Z = 1). A l'inverse, décrémenter un registre qui contient #ØØ revient à lui donner la valeur #FF, à mettre l'indicateur Négatif à 1 et l'indicateur Zéro à Ø. Il existe deux autres instructions d'incrémentation et décrémentation :

INC   Incrémente en mémoire
DEC   Décrémente en mémoire

Ces instructions permettent d'ajouter ou de retrancher 1 à la valeur contenue dans un emplacement mémoire. Par exemple :

```
INC #7Ø          \ Ajoute 1 à l'octet d'adresse #7Ø
DEC #42Ø         \ Retranche 1 à l'octet d'adresse #42Ø
```

Ces deux instructions affectent les indicateurs Négatif et Zéro dans les mêmes conditions que les instructions précédentes. Le programme 15 montre comment on peut utiliser ces instructions pour afficher la chaine «ABC» à l'écran.

Programme 15

```
 1Ø  REM * * Incrémentation d'un registre * *
 2Ø  CODE = #4ØØ
 3Ø  FOR INDEX = Ø TO 11
 4Ø    READ OCTET
 5Ø    POKE CODE + INDEX, OCTET
 6Ø  NEXT INDEX
 7Ø
 8Ø  REM * * Données du code machine * *
 9Ø  DATA #A9, #41      : REM LDA @"A"
1ØØ  DATA #85, #7Ø      : REM STA #7Ø
11Ø  DATA #AA           : REM TAX
12Ø  DATA #E8           : REM INX
13Ø  DATA #86, #71      : REM STX #71
14Ø  DATA #E8           : REM INX
15Ø  DATA #86, #72      : REM STX #72
16Ø  DATA #6Ø           : REM RTS
17Ø
18Ø  REM * * Initialisation et exécution * *
19Ø  CLS
195  CALL (CODE)
2ØØ  FOR LOC = Ø TO 2
21Ø    NB = PEEK (#7Ø + LOC)
22Ø    PRINT CHR$(NB)
23Ø  NEXT LOC
```

Les comparaisons

Il existe trois instructions de comparaison :

CMP Compare avec l'accumulateur
CPX Compare avec le registre X
CPY Compare avec le registre Y

Le contenu de chaque registre peut être comparé avec l'octet dont l'adresse est spécifiée à la suite de l'instruction, ou, comme c'est souvent le cas, avec la valeur qui est indiquée à la suite du mnémonique. Les valeurs ainsi testées ne sont pas altérées par la comparaison. Mais les indicateurs Négatif, Zéro, et de Retenue sont affectés selon le résultat de la comparaison. Comment? Tout d'abord, le 65Ø2 met l'indicateur de Retenue à 1 (C = 1), puis il retranche la valeur indiquée du contenu du registre. Si cette valeur est inférieure ou égale à celle du registre, l'indicateur de Retenue reste à 1. Si ces deux valeurs sont égales, l'indicateur Zéro est également mis à 1. Si l'indicateur de Retenue est à Ø, cela signifie que la valeur est supérieure à celle du registre, et qu'une retenue est intervenue au cours de la soustraction. En général (mais pas toujours), ce dernier cas a pour effet de mettre l'indicateur Négatif à 1. En fait cela ne se produit que si l'on compare des compléments à 2. Le tableau 12.1 récapitule ces tests.

Voici quelques exemples d'instructions de comparaison :

```
CMP @#41
CPY #42Ø, X
CPX #96ØØ
```

Tableau 12.1

Test	Indicateurs		
	C	Z	N
Le contenu du registre est inférieur à la valeur donnée	Ø	Ø	1
Le contenu du registre est égal à la valeur donnée	1	1	Ø
Le contenu du registre est supérieur à la valeur donnée	1	Ø	Ø

Les branchements

En fonction du résultat d'une comparaison on pourra, soit effectuer un branchement arrière pour répéter une boucle, soit effectuer un branchement à un autre endroit du programme, soit poursuivre normalement l'exécution. Ce type de branchement est appelé *branchement conditionnel.* Il existe huit instructions de branchement conditionnel :

BNE	Branche si Non Egal	Z = Ø
BEQ	Branche si Egal	Z = 1
BCC	Branche si indicateur de Retenue à Ø	C = Ø
BCS	Branche si indicateur de Retenue à 1	C = 1
BPL	Branche si indicateur Négatif à Ø	N = Ø
BMI	Branche si indicateur Négatif à 1	N = 1
BVC	Branche si indicateur de débordement à Ø	V = Ø
BVS	Branche si indicateur de débordement à 1	V = 1

Réécrivons maintenant en code machine le programme BASIC qui affiche 10 points d'exclamation (voir page 64) :

Programme 16

```
1Ø  REM * * 10 points d'exclamation * *
2Ø  CODE = #4ØØ
3Ø  FOR INDEX = Ø TO 12
4Ø    READ OCTET
5Ø    POKE CODE + INDEX, OCTET
6Ø  NEXT INDEX
7Ø
8Ø  REM * * Les données du code machine * *
9Ø  DATA #A2, #ØØ         : REM LDX @#ØØ
1ØØ DATA #A9, #21         : REM @"!"
11Ø DATA #9D, #AA,#BB     : REM STA #BBAA, X
12Ø DATA #E8              : REM INX
13Ø DATA #EØ, #ØA         : REM CPX @1Ø
14Ø DATA #DØ, #F8         : REM BNE -8
15Ø DATA #6Ø              : REM RTS
16Ø
17Ø REM * * Initialisation et exécution * *
18Ø CLS
19Ø CALL (CODE)
```

Les lignes 9Ø et 1ØØ initialisent le registre X et placent le code ASCII du point d'exclamation dans l'accumulateur. Après avoir rangé ce code dans la mémoire de l'écran (ligne 11Ø), on incrémente le registre X (ligne 12Ø) et on teste sa valeur en ligne 13Ø pour savoir si on a affiché les dix points d'exclamation. Si ce n'est pas le cas (X<>1Ø), l'indicateur Zéro restera à zéro et l'instruction BNE en ligne 14Ø sera exécutée, provoquant un branchement arrière en ligne 11Ø. Si l'instruction de comparaison a pour résultat «vrai», l'indicateur Zéro est mis à 1 et l'instruction RTS de la ligne 15Ø est exécutée.

On peut améliorer ce programme. Lorsqu'un registre sert *exclusivement* comme compteur de boucle, il est plus efficace d'écrire le code machine de façon à faire un compte à rebours. Pour quelle raison ? Vous vous rappelez peut-être avoir lu dans le chapitre 6 que, lorsqu'un registre contient zéro après une décrémentation, l'indicateur Zéro est mis à 1. Partant de ce principe on peut se dispenser de l'instruction CPX @1Ø et réécrire le programme comme suit :

Programme 17

```
1Ø  REM * * Compte à rebours * *
2Ø  CODE = #4ØØ
```

```
30  FOR INDEX = 0 TO 12
40    READ OCTET
50    POKE CODE + INDEX, OCTET
60  NEXT INDEX
70
80  REM * * Les données du code machine * *
90  DATA #A2, #0A       : REM LDX @10
100 DATA #A9, #21       : REM LDA @"!"
110 DATA #9D, #AA, #BB: REM STA #BBAA, X
120 DATA #CA            : REM DEX
140 DATA #D0, #FA       : REM BNE -6
150 DATA #60            : REM RTS
160
170 REM * * Initialisation et exécution * *
180 CLS
190 CALL (CODE)
```

Si vous regardez ce listing en détail, vous verrez en ligne 140 que le code opérateur de l'instruction BNE est suivi d'un seul octet, et non d'une adresse. On appelle cet octet un octet de *déplacement*, et ce mode d'adressage *l'adressage relatif.* L'opérande, ici #FA, indique au microprocesseur qu'il doit effectuer un branchement arrière, de six octets. Pour distinguer les branchements arrière des branchements avant, on utilise le *binaire signé.* Un nombre négatif signale un branchement arrière et un nombre positif indique un branchement avant. Il est important de savoir calculer les déplacements ; essayons donc sur un exemple.

Avant de vous installer au clavier de votre Oric il vaut mieux, d'une manière générale, préparer sur papier votre code machine. Bien qu'il soit tout à fait possible d'écrire directement au clavier, les erreurs de codage peuvent poser de gros problèmes (et je sais de quoi je parle!). Pour rendre vos boucles et vos branchements plus lisibles, vous pouvez par exemple utiliser des *étiquettes.* Dans le programme 17, vous pouvez par exemple utiliser l'étiquette BOUCLE pour marquer le point de branchement, comme le montre le tableau 12.1.

Tableau 12.1

Etiquette	Mnémoniques	Commentaires
DEBUT		Début du code
	LDX @10	Compteur de la boucle
	LDA @ "!"	Code ASCII pour "!"
BOUCLE		Point de branchement
	STA #BBAA, X	Range dans la mémoire de l'écran
	DEX	X = X − 1
	BNE BOUCLE	Continue jusqu'à ce que X = 0
	RTS	C'est fini!

A ce stade, il est possible d'élaborer le code machine correspondant et de l'insérer dans le tableau. Pour calculer le déplacement du branchement, il suffit de compter le nombre d'octets à partir de l'octet de déplacement en remontant jusqu'à l'étiquette BOUCLE.

```
BOUCLE
          STA #BBAA, X      3 octets
          DEX               1 octet
          BNE BOUCLE        2 octets
```

On obtient un total de 6 octets. Notez que le déplacement relatif et le code opérateur de BNE sont inclus dans ce nombre, puisque le Compteur de Programme contiendra l'adresse de l'instruction *qui suit* le branchement. Pour convertir ce branchement arrière en un nombre en binaire signé, il suffit de calculer son complément à deux (reportez-vous au chapitre 3, si vous ne vous en souvenez pas).

```
  Ø Ø Ø Ø Ø 1 1 Ø    (6)
  1 1 1 1 1 Ø Ø 1
+               1
  ---------------
  1 1 1 1 1 Ø 1 Ø    (– 6= #FA)
```

Comme les instructions de branchement occupent toujours deux octets, des déplacements effectifs de – 126 octets (– 128 + 2) et de + 129 octets (+ 127 + 2) sont possibles.

Pour voir un exemple de branchement conditionnel avant, entrez et faites tourner le programme suivant qui affiche un «O» si l'adresse #4ØØ contient Ø, et un «N» dans les autres cas.

Programme 18

```
 1Ø  REM * * Branchement avant * *
 2Ø  REM * * Initialiser HIMEM #96ØØ * *
 3Ø  CODE = #96ØØ
 4Ø  FOR INDEX = Ø TO 16
 5Ø    READ OCTET
 6Ø    POKE CODE + INDEX, OCTET
 7Ø  NEXT INDEX
 8Ø
 9Ø  REM * * Données du code machine * *
1ØØ  DATA #AD, #ØØ, #Ø4 : REM LDA #4ØØ
11Ø  DATA #FØ, #Ø6      : REM BEQ +6
12Ø  DATA #A9, #4E      : REM LDA @"N"
13Ø  DATA #8D, #AA, #BB: REM STA #BBAA
14Ø  DATA #6Ø           : REM RTS
15Ø  DATA #A9, #4F      : REM LDA @"O"
16Ø  DATA #8D, #AA, #BB: REM STA #BBAA
```

```
170  DATA #60              : REM RTS
180
190  REM * * Initialisation et exécution * *
200  CLS
210  CALL (CODE)
```

A l'exécution, ce programme commence par charger l'octet d'adresse #400 dans l'accumulateur (ligne 100). Si cet octet contient #00, l'indicateur Zéro est mis à 1, donc l'instruction BEQ de la ligne 110 est exécutée et les octets provoquant l'écriture d'un «N» dans la mémoire de l'écran ne sont pas pris en compte. Comme il s'agit d'un branchement avant; on utilise un nombre positif (c'est à dire inférieur à #80) pour indiquer le déplacement. Le tableau 12.2 vous montre comment j'ai utilisé les labels pour calculer ce déplacement.

Tableau 12.2

Label	Mnémoniques	Commentaires
DEBUT		A partir de #9600
	LDA #400	Charge l'octet
	BEQ ZERO	Branche si Z = 0
	LDA @ "N"	Code ASCII de N
	STA #BBAA	L'écrit dans la mémoire de l'écran
	RTS	Retour sous Basic
ZERO	LDY @ "O"	Code ASCII de O
	STA #BBAA	L'écrit dans la mémoire de l'écran
	RTS	Retour sous Basic

Si on compte les octets qui suivent le branchement jusqu'au label ZERO, on obtient :

```
        BEQ ZERO
        LDA @ "N"     2 octets
        STA #BBAA     3 octets
        RTS           1 octet
ZERO
```

Soit un total de 6 octets. Vous trouverez dans l'annexe 4 deux tables qui peuvent vous aider à convertir le nombre d'octets en une valeur de déplacement correcte. Cela vous évitera d'avoir à faire le calcul vous-même.

Les compteurs en mémoire

Les programmes qui travaillent avec des adresses absolues, auront systématiquement besoin de routines qui incrémentent ou décrémentent les valeurs de ces adresses. Un exemple typique sera un programme qui utilise l'adressage post-indexé indirect pour accéder séquentiellement à une suite d'octets consécutifs. Les deux programmes suivants illustrent cette façon de faire. Dans le premier cas, on incrémente les adresses mémoire.

Programme 19

```
 1Ø  REM * * Incrémentation en mémoire * *
 2Ø  REM * * Réinitialiser HIMEM #96ØØ * *
 3Ø  CODE = #96ØØ
 4Ø  COMPTEUR = #4ØØ
 5Ø  DOKE COMPTEUR, Ø
 6Ø  FOR INDEX = Ø TO 8
 7Ø    READ OCTET
 8Ø    POKE CODE + INDEX, OCTET
 9Ø  NEXT INDEX
1ØØ
11Ø  REM * * Les données du code machine * *
12Ø  DATA #EE, #ØØ, #Ø4  : REM INC #4ØØ
13Ø  DATA #DØ, #Ø3       : REM BNE +3
14Ø  DATA #EE, #Ø1, #Ø4  : REM INC #4Ø1
15Ø  DATA #6Ø            : REM RTS
16Ø
17Ø  REM * * Initialisation et exécution * *
18Ø  REPEAT
19Ø    PRINT DEEK (COMPTEUR)
2ØØ    CALL (CODE)
21Ø    GET A$
22Ø  UNTIL A$ = "S"
```

Les lignes 12Ø à 14Ø contiennent le code machine. A chaque exécution de la routine, l'octet bas, en #4ØØ, est incrémenté. Toutes les fois qu'il passe de #FF à #ØØ, l'indicateur Zéro est mis à 1 et le branchement (en ligne 13Ø) ne s'effectue pas. Par conséquent, c'est l'octet haut du compteur, en #4Ø1, qui sera incrémenté. La décrémentation d'un compteur est un peu moins simple à réaliser.

Programme 20

```
1Ø  REM * * Décrémentation en mémoire * *
2Ø  REM * * Réinitialiser HIMEM #96ØØ * *
3Ø  CODE = #96ØØ
4Ø  COMPTEUR = #4ØØ
5Ø  DOKE COMPTEUR, #FFFF
6Ø  FOR INDEX = Ø TO 11
```

```
 7Ø     READ OCTET
 8Ø     POKE CODE + INDEX, OCTET
 9Ø   NEXT INDEX
1ØØ
11Ø   REM * * Les données du code machine * *
12Ø   DATA #AD, #ØØ, #Ø4  : REM LDA #4ØØ
13Ø   DATA #DØ, #Ø3       : REM BNE +3
14Ø   DATA #CE, #Ø1, #Ø4  : REM DEC #4Ø1
15Ø   DATA #CE, #ØØ, #Ø4  : REM DEC #4ØØ
16Ø   DATA #6Ø            : REM RTS
17Ø
18Ø   REM * * Initialisation et exécution * *
19Ø   REPEAT
2ØØ     PRINT DEEK (COMPTEUR)
21Ø     CALL (CODE)
22Ø     GET A$
23Ø   UNTIL A$ = "S"
```

Tout d'abord on charge l'accumulateur avec l'octet bas du compteur, #4ØØ (en ligne 12Ø), ce qui affecte l'indicateur Zéro. S'il est à 1, l'octet bas contient #ØØ, il faut donc décrémenter l'octet haut (ligne 14Ø), sinon le branchement s'effectue et seul l'octet bas du compteur sera décrémenté. On aurait très bien pu utiliser l'un des registres d'index à la place de l'accumulateur (en ligne 12Ø).Si tous les registres sont utilisés, on peut aussi écrire :

```
            INC #4ØØ
            DEC #4ØØ
            BNE DECBAS
            DEC #4Ø1
DECBAS      DEC #4ØØ
```

Le fait d'incrémenter, puis de décrémenter l'octet bas de #4ØØ affectera l'indicateur Zéro dans les mêmes conditions qu'une instruction de chargement.

Chapitre 13

Les sous-programmes et les sauts

Les sous-programmes

Si vous connaissez les instructions GOSUB et RETURN du BASIC, vous n'aurez aucune difficulté à comprendre leurs équivalents en assembleur :

JSR appel d'un sous-programme
RTS retour du sous-programme

Si cela ne vous est pas familier, voici une explication : il est fréquent de constater, en écrivant un programme, la nécessité d'effectuer à plusieurs reprises la même opération. Plutôt que de recopier le même groupe de mnémoniques chaque fois, ce qui serait une perte de temps et allongerait exagérément le programme, il est possible de les écrire une seule fois, en dehors du programme principal, et de les appeler en cas de besoin. Toutes les séquences répétitives en assembleur ne justifient toutefois pas la création d'un sous-programme. Par exemple :

INX : DEY : STA #22

est très fréquent, mais n'occupe que quatre octets, c'est-à-dire la même place en mémoire qu'un appel à un sous-programme et un retour sous BASIC (JSR ... RTS). On ne gagnera donc rien à écrire un sous-programme dans ce cas, au contraire cela ralentirait l'exécution de quelques millionièmes de secondes! D'un autre côté :

CLC : LDA #22 : ADC @#12 : STA #23

peut justifier la création d'un sous-programme, étant donné qu'en utilisant l'adressage absolu, cette séquence occupera jusqu'à 1Ø octets. Regardons maintenant un petit programme qui contient plusieurs appels à des sous-programmes.

Programme 21

```
1Ø  REM * * Appel de sous-programmes * *
2Ø  REM * * Réinitialiser HIMEM #96ØØ * *
3Ø  CODE = #96ØØ
4Ø  SPROG = #4ØØ
5Ø  REM * * Installation du sous-programme * *
6Ø  FOR INDEX = Ø TO 3
7Ø    READ OCTET
8Ø    POKE SPROG + INDEX, OCTET
9Ø  NEXT INDEX
1ØØ REM * * Installation du programme principal * *
11Ø FOR INDEX = Ø TO 15
12Ø   READ OCTET
13Ø   POKE CODE + INDEX, OCTET
14Ø NEXT INDEX
15Ø
16Ø REM * * Les données du code machine * *
17Ø REM * * Sous-programme * *
18Ø DATA #9D, #AA, #BB    : REM STA #BBAA, X
19Ø DATA #6Ø              : REM RTS
2ØØ REM * * Programme principal * *
21Ø DATA #A9, #41         : REM LDA @"A"
22Ø DATA #AB              : REM TAY
23Ø DATA #A2, #ØØ         : REM LDX @ØØ
24Ø DATA #2Ø, #ØØ, #Ø4    : REM JSR #4ØØ
25Ø DATA #E8              : REM INX
26Ø DATA #C8              : REM INY
27Ø DATA #98              : REM TYA
28Ø DATA #EØ, #1A         : REM CPX @26
29Ø DATA #DØ, #F6         : REM BNE -1Ø
3ØØ DATA #6Ø              : REM RTS
31Ø
32Ø REM * * Initialisation et exécution * *
33Ø CLS
34Ø CALL (CODE)
```

Le sous-programme est écrit dans la zone de la mémoire réservée au code machine de l'utilisateur. Comme il n'occupe que quatre octets, il aurait été bien plus efficace de l'inclure dans le programme principal. Mais pour les besoins de la démonstration, j'ai délibérément choisi un sous-programme très court. Le programme principal est rangé à partir de #96ØØ, au-dessus de HIMEM préalablement réinitialisé. Il utilise l'adressage indexé absolu pour afficher l'alphabet à l'écran. On charge tout d'abord le code ASCII de la lettre «A» dans l'accumulateur (ligne 21Ø), puis on le transfère dans le registre Y avec l'instruction de transfert appropriée (ligne 22Ø). Cette étape est importante, puisqu'il va falloir incrémenter le code ASCII contenu dans le registre pour obtenir le code de la lettre suivante, et qu'il n'existe pas d'instruction d'incrémentation directe pour l'accumulateur (bien qu'on puisse éventuellement utiliser CLC, ADC @1).

Le registre X, qui sert d'index, est initialisé à Ø (ligne 23Ø) et on effectue le premier des 26 appels du sous-programme (ligne 24Ø). Le sous-programme, situé plus bas dans la carte mémoire, en #4ØØ, se charge de placer le code ASCII, et donc le caractère, dans la mémoire de l'écran (ligne 18Ø). L'instruction RTS (ligne 19Ø) redonne le contrôle au programme appelant, et *non* à BASIC. Les deux registres d'index sont incrémentés (lignes 25Ø et 26Ø), le code ASCII du caractère suivant est transféré dans l'accumulateur (ligne 27Ø) et le programme est répété 25 fois (lignes 28Ø et 29Ø).

Après ce bref aperçu sur les appels de sous-programmes et leurs fonctions, voyons concrètement comment cela se passe. Les deux instructions JSR et RTS remplissent, à elles deux, trois fonctions. Il faut d'abord sauvegarder le contenu courant du Compteur de Programme, de façon à pouvoir redonner le contrôle au programme principal le moment venu. Ensuite, il faut dire au 65Ø2 d'exécuter le sous-programme auquel on l'a envoyé. Enfin, il faut redonner la main au programme principal.

L'instruction JSR s'acquitte des deux premières tâches. Elle sauve l'adresse de retour du sous-programme en empilant les deux octets contenus dans le Compteur de Programme. Celui-ci contient, à cette étape, l'adresse du troisième octet de l'instruction JSR. Ensuite, le Compteur de Programme est chargé avec l'adresse spécifiée par JSR, ce qui a pour effet de lancer l'exécution du sous-programme.

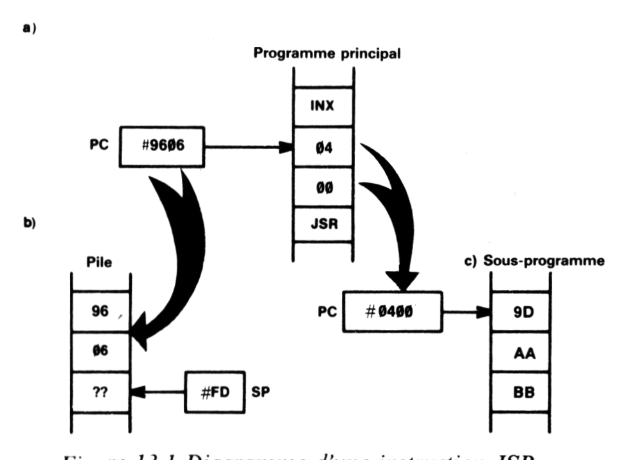


Figure 13.1 Diagramme d'une instruction JSR

La figure 13.1 montre comment s'effectuent ces opérations, et comment, en particulier, elles utilisent la pile. Lorsque le 65Ø2 rencontre une instruction JSR, le Compteur de Programme pointe sur le second octet de l'opérande (figure 13.1a). Le microprocesseur empile le contenu du Compteur de Programme en commençant par l'octet de poids fort (figure 13.1b), puis transfère l'adresse du sous-programme dans le Compteur de Programme (figure 13.1c).

En fin de sous-programme, l'instruction RTS provoque l'exécution des opérations inverses. L'adresse de retour au programme principal est dépilée et incrémentée (figure 13.2) au moment où elle est rechargée dans le Compteur de Programme pour pointer sur l'instruction suivant l'appel du sous-programme.

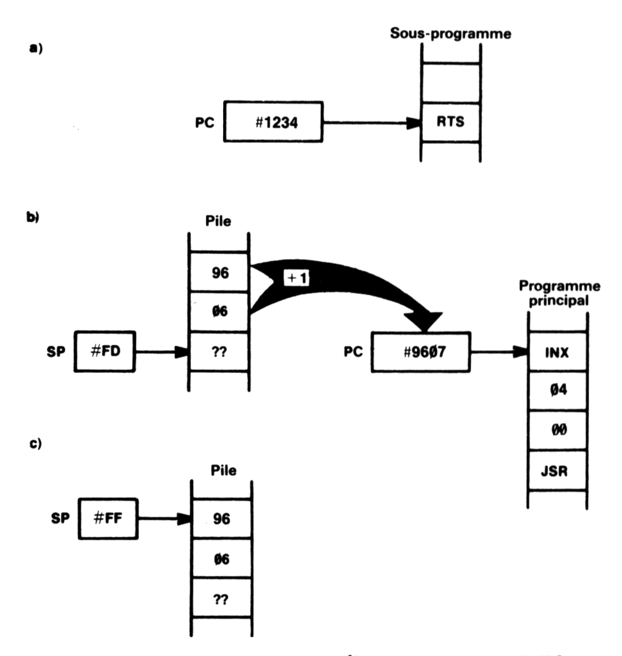


Figure 13.2 Diagramme d'une instruction RTS

Le passage de paramètres

Neuf fois sur dix, un sous-programme devra travailler sur des données définies dans le programme principal, que celui-ci doit donc lui communiquer. On dispose de trois moyens pour transmettre des paramètres, ou informations, à un sous-programme :

1. En passant par les registres.
2. En passant par la mémoire.
3. En passant par la pile.

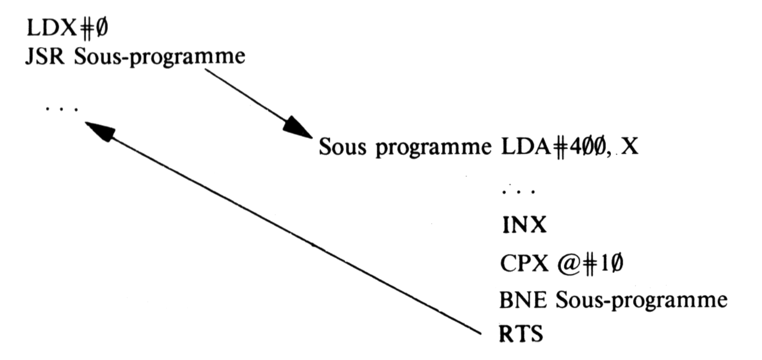
Examinons chacun de ces trois cas.

Passage par les registres

C'est évidemment la méthode la plus simple, en particulier parce qu'elle permet au sous-programme d'être indépendant de l'implantation des données en mémoire. Mais, comme on ne dispose que de trois registres, on ne peut transmettre que trois octets d'information. Les registres pouvant contenir des données importantes pour le programme principal, il faudra prendre soin de les sauvegarder d'abord, par exemple sur la pile, pour les retrouver par la suite.

Passage par la mémoire

C'est certainement la méthode la plus facile si l'on a à transmettre de nombreux paramètres à un sous-programme. Il est plus efficace d'utiliser les emplacements d'adresse #4ØØ à #42Ø inclus, puisqu'ils sont réservés à l'utilisateur. Si le sous-programme utilise plusieurs octets en mémoire, on aura intérêt à charger le registre X avec l'adresse du premier, puis à utiliser l'adressage indexé absolu en indiquant #4ØØ comme adresse de base. Par exemple :



L'inconvénient du passage de paramètres par la mémoire est la liaison du sous-programme à une zone de la mémoire. Mais, dans la plupart des cas, cela n'a pas d'importance.

Passage par la pile

Cette méthode exige des précautions, puisque le sommet de la pile contient l'adresse de retour du sous-programme. Elle nécessite deux octets en mémoire, où l'on rangera l'adresse de retour après l'avoir dépilée (bien qu'on puisse également utiliser les registres d'index). Si l'on transmet des paramètres en passant par la pile, le sous-programme doit commencer par la séquence suivante :

```
PLA             \ dépile l'octet bas de l'adresse de retour
STA ADR         \ le range en mémoire
PLA             \ dépile l'octet haut
STA ADR + 1     \ le range en mémoire.
```

Les instructions STA peuvent être remplacées par TAX et TAY si l'on préfère stocker l'adresse de retour dans les registres d'index, de manière à charger l'octet bas dans le registre X et l'octet haut dans le registre Y. Après avoir dépilé les paramètres, on peut empiler à nouveau l'adresse de retour :

```
LDA ADR + 1
PHA
LDA ADR
PHA
```

Rappelez-vous que la pile a une structure de type LIFO et qu'il faut, par conséquent, dépiler les octets dans l'ordre inverse de celui dans lequel on les a empilés pour les sauvegarder. Quand on transmet un nombre variable de paramètres à un sous-programme, on peut en déterminer le nombre à chaque exécution en testant le contenu du Pointeur de Pile. Pour cela, on transfère sa valeur dans le registre X avec l'instruction TSX, et on incrémente X à chaque dépilement jusqu'à ce qu'il soit égal à #FF, par exemple, ce qui indique que la pile est vide. Pour déterminer la valeur limite du test, il faut savoir si le sous-programme en cours d'exécution a été appelé par un autre sous-programme. Dans ce cas, on dit qu'il est imbriqué. La valeur #FF est donc ici une simple supposition selon laquelle il n'y a que des paramètres dans la pile.

Les sauts

Une instruction JMP agit un peu comme une instruction GOTO en BASIC, c'est-à-dire en faisant exécuter une autre partie du programme. En code machine, l'opérande désigne une adresse absolue, et non une ligne de programme (cette notion n'ayant pas de sens en code machine). Cette instruction charge simplement les deux octets de l'adresse spécifiée après le code opérateur dans le Compteur de Programme, provoquant ainsi un saut.

On utilise en général une instruction JMP pour sauter une suite d'instructions qu'on ne veut pas faire exécuter lorsqu'une condition n'a pas été vérifiée. Par exemple :

```
            BCC SUITE
            JMP AILLEURS
SUITE       LDA OCTET
            ASL A
            INX
            DEY
AILLEURS    STA TEMP
```

Dans ce cas, si l'indicateur de Retenue est à Ø, l'instruction JMP n'est pas prise en compte et les instructions à partir de l'étiquette SUITE sont exécutées. Si l'indicateur de Retenue est à 1, la condition n'est pas vérifiée et le branchement sur SUITE ne se fait pas, l'instruction JMP est prise en compte ignorant ainsi toutes les instructions jusqu'à l'étiquette AILLEURS.

Nous avions vu, au cours du chapitre 10, une autre utilisation de JMP, le « saut indirect ». Dans ce cas, l'adresse à laquelle saute effectivement le programme est stockée dans un vecteur dont l'adresse est spécifiée après le mnémonique. JMP (#42Ø) en est un exemple.

Chapitre 14

Les décalages et les rotations

En un mot, ces instructions permettent de décaler les bits d'un octet d'un rang vers la gauche ou d'un rang vers la droite. Il existe quatre instructions de ce type :

ASL Décalage arithmétique à gauche
LSR Décalage logique à droite
ROL Rotation à gauche
ROR Rotation à droite.

Toutes ces instructions peuvent agir directement sur l'accumulateur ou sur un octet en mémoire :

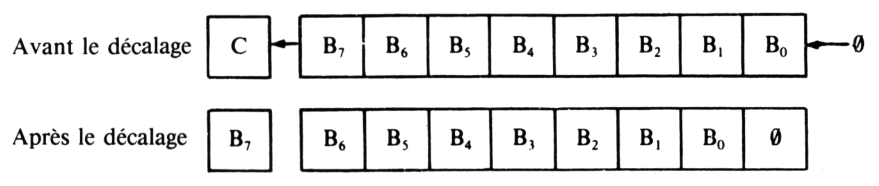
ASL A \ Décalage arithmétique à gauche de l'octet contenu dans l'accumulateur

ROL #7Ø \ Rotation à gauche de l'octet d'adresse #7Ø.

Examinons chacune de ces instructions plus en détail.

Le décalage arithmétique à gauche

ASL décale tous les bits d'un octet d'un bit vers la gauche.



Le bit 7 (B_7) est décalé dans l'indicateur de Retenue, et un « Ø» est placé dans le bit Ø, les bits 1 à 6 sont tous décalés d'un rang vers la gauche. Cette instruction a pour effet principal de doubler la valeur de l'octet.

Exemple :

		C	Accumulateur
LDA @#42	\ charge #42 dans l'accumulateur	X	Ø 1 Ø Ø Ø Ø 1 Ø
ASL A	\ décale l'accumulateur vers la gauche	Ø	1 Ø Ø Ø Ø 1 Ø Ø

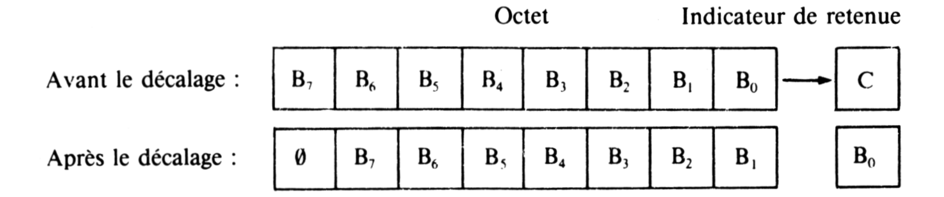
L'accumulateur contient maintenant #84, le double de la valeur initiale! Le programme 22, qui lit un nombre (inférieur à 64), le multiplie par quatre en exécutant deux fois ASL A, et affiche le résultat, est un autre exemple d'utilisation de l'instruction ASL.

Programme 22

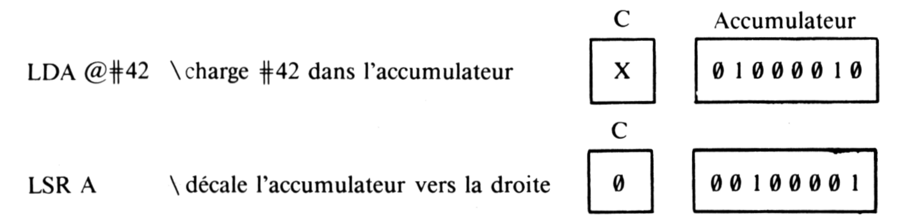
```
 1Ø  REM * * Multiplication par quatre * *
 2Ø  CODE = #4ØØ
 3Ø  FOR INDEX = Ø TO 8
 4Ø    READ OCTET
 5Ø    POKE CODE + INDEX, OCTET
 6Ø  NEXT INDEX
 7Ø
 8Ø  REM * * Données du code machine * *
 9Ø  DATA #AD, #2Ø, #Ø4  : REM LDA #42Ø
1ØØ  DATA #ØA            : REM ASL A
11Ø  DATA #ØA            : REM ASL A
12Ø  DATA #8D, #2Ø, #Ø4  : REM STA #42Ø
13Ø  DATA #6Ø            : REM RTS
14Ø
15Ø  REM * * Initialisation et exécution * *
16Ø  CLS
17Ø  INPUT «Nombre à multiplier : »; NB
18Ø  POKE #42Ø, NB
19Ø  CALL (CODE)
2ØØ  PRINT "Le résultat est :";
21Ø  PRINT PEEK (#42Ø)
```

Le décalage logique à droite

LSR est comparable à ASL, si ce n'est que les bits sont décalés vers la droite. Le bit Ø (B_0) passe dans l'indicateur de Retenue et un Ø est placé dans le bit libéré par le bit 7 (B_7).



Cette instruction aurait aussi bien pu s'appeler décalage arithmétique à droite, puisqu'elle divise effectivement la valeur de l'octet par deux. Par exemple :



L'accumulateur contient maintenant #21, c'est-à-dire la moitié de la valeur initiale. Faire :

LSR A :BCS Adresse 1
ou :
LSR A :BCC Adresse 2

est un bon moyen pour tester le bit Ø de l'accumulateur.
Le programme 23 teste l'état du bit Ø d'un caractère ASCII saisi au clavier, en le décalant dans l'indicateur de Retenue. Si celui-ci est à Ø, le programme affiche un Ø à l'écran, sinon un 1.

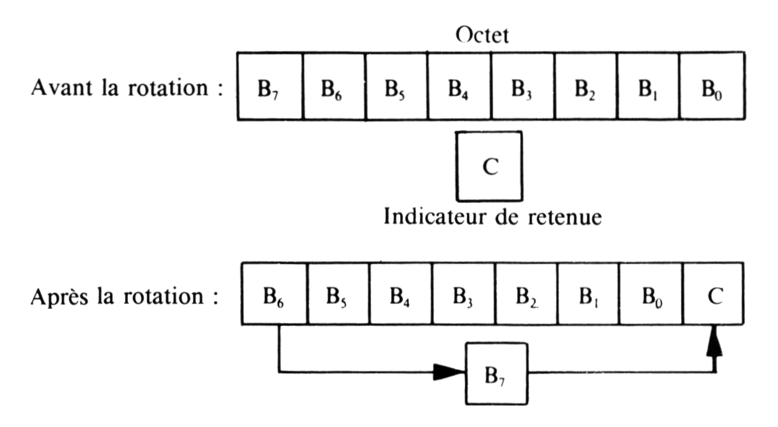
Programme 23

```
 1Ø  REM * * Test du bit Ø * *
 2Ø  CODE = #4ØØ
 3Ø  FOR INDEX = Ø TO 11
 4Ø    READ OCTET
 5Ø    POKE CODE + INDEX, OCTET
 6Ø  NEXT INDEX
 7Ø
 8Ø  REM * * Les données du code machine * *
 9Ø  DATA #AD, #2Ø, #Ø4  : REM LDA #42Ø
1ØØ  DATA #4A            : REM LSR A
11Ø  DATA #A9, #ØØ       : REM LDA @Ø
12Ø  DATA #69, ØØ        : REM ADC @Ø
```

```
130 DATA #8D, #20, #04  : REM STA #420
140 DATA #60            : REM RTS
150
160 REM * * Initialisation et exécution * *
170 CLS
180 INPUT "Nombre à tester :"; NB
190 POKE #420, NB
200 CALL (CODE)
210 PRINT PEEK (#420)
```

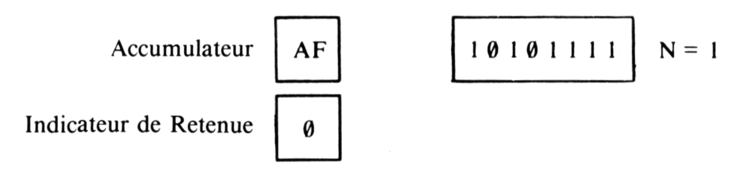
La rotation à gauche

Cette instruction se sert de l'indicateur de Retenue comme d'un neuvième bit, en faisant tourner tous les bits de l'octet d'un rang vers la gauche. Le bit 7 va donc se retrouver dans l'indicateur de Retenue, dont le contenu a pris la place du bit 0.

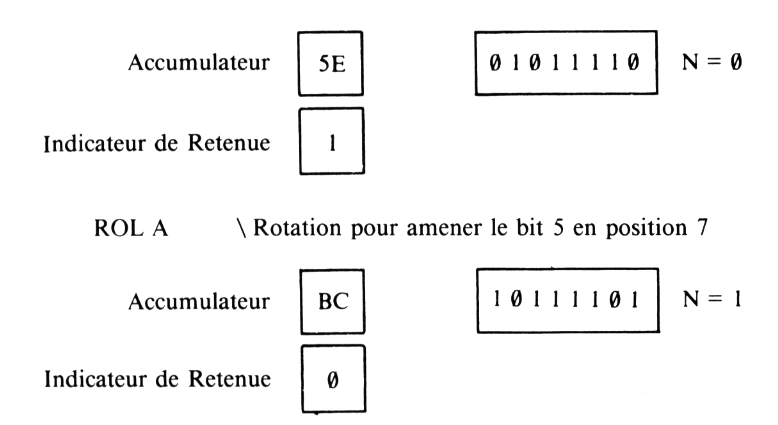


L'instruction ROL est très pratique pour tester les bits du quartet haut de l'accumulateur. On amène par rotations successives le bit à tester en position 7, pour donner sa valeur à l'indicateur Négatif.

Exemple : Test du bit 5 de l'accumulateur.



ROL A \Rotation pour amener le bit 6 en position 7

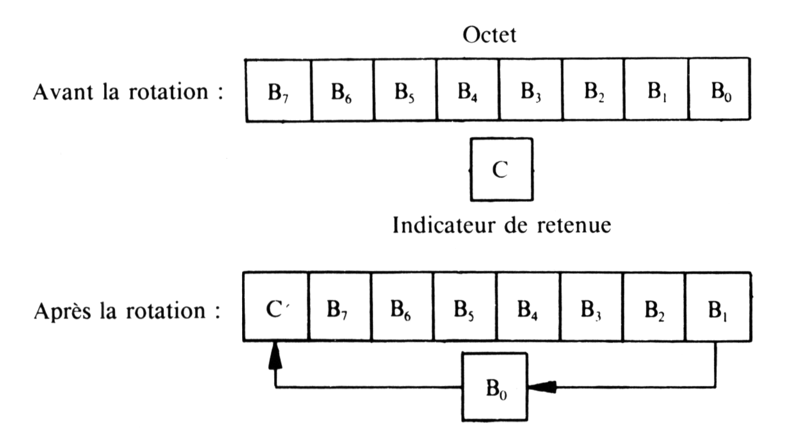


ROL A   \ Rotation pour amener le bit 5 en position 7

L'indicateur Négatif est maintenant à 1, ce qui signifie que le bit 5 de l'accumulateur était à 1.

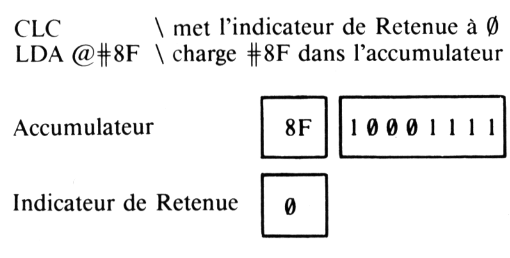
La rotation à droite

Cette instruction agit comme ROL, mais ici les bits se déplaçent vers la droite.



Exemple : Appliquer ROR à l'accumulateur chargé avec #8F

```
CLC          \ met l'indicateur de Retenue à Ø
LDA @#8F     \ charge #8F dans l'accumulateur
```



ROR   \rotation à droite

Accumulateur	47	∅ 1 ∅ ∅ ∅ 1 1 1
Indicateur de Retenue	1	

Un peu de logique

Si on a besoin d'appliquer l'une des instructions précédentes *plusieurs fois* à un octet en mémoire, il est préférable de charger la donnée dans l'accumulateur pour la manipuler, puis de la ranger à nouveau en mémoire, plutôt que de travailler directement en mémoire. Par exemple, pour effectuer quatre rotations à droite sur le contenu de l'adresse #1234, on pourrait écrire :

```
ROR #1234
ROR #1234
ROR #1234
ROR #1234
```

On utilise pour cela 12 octets en mémoire, 4 pour les instructions et 8 pour les adresses. On obtient le même résultat en écrivant :

```
LDA #1234
ROR A
ROR A
ROR A
ROR A
STA #1234
```

Dans ce cas, on utilise deux octets de moins et on gagne en rapidité d'exécution. Jusqu'ici nous n'avons parlé que de décalages et de rotations sur un seul octet. En combinant ces instructions, on peut effectuer toutes ces opérations sur des valeurs de deux octets, comme par exemple #∅123. Pour décaler à gauche tous les bits d'une donnée sur deux octets, respectivement d'adresse HAUT et BAS, on utilisera conjointement ASL et ROL :

```
ASL BAS        \ décale l'octet BAS et met le bit 7 dans l'indicateur
                 de Retenue
ROL HAUT       \ l'amène par rotation dans le bit ∅ de l'octet HAUT.
```

On peut effectuer de même un décalage à droite sur deux octets, en écrivant :

```
LSR HAUT       \ décale l'octet HAUT et met le bit ∅ dans l'indi-
                 cateur de Retenue
ROR BAS        \ l'amène par rotation dans le bit 7 de l'octet
                 BAS.
```

Notez que les octets sont manipulés dans l'ordre inverse, puisque nous voulons déplacer les bits en sens inverse. Comme nous l'avons vu pour des valeurs sur un seul octet, les valeurs sur deux octets peuvent être multipliées ou divisées par deux. Pour effectuer une rotation sur deux octets, il suffit de faire deux fois une rotation simple. Mais comme pour les décalages, il est capital de le faire dans le bon ordre. Pour appliquer ROR sur deux octets, on écrit :

```
ROR HAUT            \ amène le bit ∅ de l'octet HAUT dans l'indica-
                      teur de Retenue
ROR BAS             \ puis le fait passer dans le bit 7 de l'octet BAS
```

Alors que pour effectuer ROL sur deux octets, on écrira :

```
ROL BAS             \ fait passer le bit 7 de l'octet BAS dans l'indica-
                      teur de Retenue
ROL HAUT            \ puis dans le bit ∅ de l'octet HAUT.
```

Enfin, pour en revenir aux décalages sur un seul octet, il est possible de décaler le contenu de l'accumulateur vers la droite, tout en sauvegardant le bit de signe en utilisant la technique suivante :

```
TAY                 \ sauvegarde le contenu de l'accumulateur dans le
                      registre Y
ASL A               \ fait passer le bit de signe (bit 7) dans l'indicateur
                      de Retenue
TYA                 \ replace la valeur initiale dans l'accumulateur
ROR A               \ replace le bit de signe dans le bit 7.
```

Le registre Y a servi de stockage temporaire. On aurait aussi bien pu utiliser le registre X ou un emplacement en mémoire.

Afficher du binaire

Il est souvent nécessaire de connaître sous forme binaire le contenu d'un registre ou d'un octet en mémoire. C'est particulièrement vrai pour le registre d'Etat où chaque indicateur peut fournir des informations précieuses sur la manière dont un programme s'exécute. Le programme 24 affiche la représentation binaire d'un octet. Il le fait, à titre d'exemple, pour le contenu du registre d'Etat au moment où le programme s'exécute.

Programme 24

```
 1∅  REM * * Affichage en binaire du registre d'Etat * *
 15  REM * * HIMEM #96∅∅ * *
 2∅  CODE = #96∅∅
 3∅  FOR INDEX = ∅ TO 2∅
 4∅    READ OCTET
 5∅    POKE CODE + INDEX, OCTET
 6∅  NEXT INDEX
 7∅
 8∅  REM * * Les données du code machine * *
 9∅  DATA #∅8              : REM PHP
1∅∅  DATA #68              : REM PLA
11∅  DATA #8D, #∅∅, #∅4    : REM STA #4∅∅
```

```
120  DATA #A2, #08          : REM LDX @#8
130  DATA #0E, #00, #04     : REM ASL #400
140  DATA #A9, #00          : REM LDA @#0
150  DATA #69, #00          : REM ADC @#0
160  DATA #9D, #00, #04     : REM STA #400, X
170  DATA #CA               : REM DEX
180  DATA #D0, #F3          : REM BNE - 13
190  DATA #60               : REM RTS
200
210  CLS
220  PRINT "Registre d'Etat"
230  "N V - B D I Z C"
240  CALL (CODE)
250  FOR INDEX = 8 TO 1 STEP - 1
260    PRINT PEEK (#400 + INDEX);
270  NEXT INDEX
```

Pour pouvoir être manipulé, le contenu du registre d'Etat doit être sauvegardé en mémoire. Pour cela on commence par le transférer dans l'accumulateur, en l'empilant d'abord (PHP en ligne 9Ø), puis en le dépilant dans l'accumulateur (PLA en ligne 1ØØ); on le range enfin dans la zone utilisateur (ligne 11Ø). On utilise le registre X pour compter les 8 bits de l'octet en l'initialisant en conséquence (ligne 12Ø). Par un décalage arithmétique à gauche en ligne 13Ø, on fait passer le bit le plus significatif de l'octet d'adresse #4ØØ dans l'indicateur de Retenue. On charge #ØØ dans l'accumulateur (ligne 14Ø) et on y ajoute Ø avec l'instruction ADC (ligne 15Ø). Non, je ne suis pas en plein délire! Rappelez-vous que l'opération ADC prend en compte le contenu de l'indicateur de retenue. Si ce dernier est à 1 après le décalage, l'accumulateur va contenir 1 (sinon il ne sera pas modifié). Dans tous les cas le résultat est sauvegardé dans la zone utilisateur (ligne 16Ø) par adressage indexé. Comme le registre X avait été initialisé à 8 et sert en même temps de registre d'index, les résultats des décalages sont stockés dans l'ordre inverse. Cette séquence est exécutée 8 fois et le registre X est décrémenté à chaque exécution jusqu'à ce qu'il contienne zéro. Après être revenu sous BASIC, on affiche la valeur de chaque indicateur en l'extrayant de la zone utilisateur au moyen d'une boucle comportant un STEP négatif.

Pour vérifier que le programme affiche réellement la valeur des différents indicateurs, on peut y inclure une ligne qui les modifie à dessein. Par exemple :

```
85  DATA #A9, #FF       : REM LDA @#FF     N = 1 et Z = 0
```

ou :

```
85  DATA #A9,# 00       : REM LDA @#00     N = 0 et Z = 1
```

Ces instructions affecteront les indicateurs dans les conditions mentionnées en REM. N'oubliez pas de modifier le nombre de répétitions de la boucle de lecture des DATA en y ajoutant le nombre d'octets que vous insérez (deux dans l'exemple ci-dessus).

BIT

L'instruction BIT permet de tester les bits d'un octet spécifié. Sa principale caractéristique est qu'elle *n'*altère *pas* l'octet testé, mais qu'elle affecte, comme vous l'avez deviné, plusieurs indicateurs du registre d'Etat. Ainsi :

1. L'indicateur Négatif est chargé avec la valeur du bit 7 de l'octet testé.
2. L'indicateur de débordement (V) est chargé avec la valeur du bit 6 de l'octet testé.
3. L'indicateur Zéro est à un si le résultat de l'opération AND entre l'accumulateur et l'octet testé est égal à zéro.

En chargeant dans l'accumulateur un octet de « masquage », il est possible de tester n'importe quel bit d'un octet en mémoire. Par exemple, pour tester l'octet d'adresse TEMP, afin de savoir si le bit Ø est à Ø, on peut écrire :

```
LDA @#1                 \ ØØØØØØØ1
BIT TEMP                \ test du bit Ø
```

Si le bit Ø de TEMP est à Ø, l'indicateur Zéro est à 1, sinon il reste à Ø, ce qui permet d'utiliser BNE et BEQ dans de nombreuses situations de test. La technique du masquage ne sera utile que pour tester les bits Ø à 5, puisque les bits 6 et 7 sont directement recopiés dans les indicateurs Négatif et de débordement, qui ont leurs propres instructions de test.

```
BIT TEMP
BMI                     \ branche si le bit 7 est à 1
BPL                     \ branche si le bit 7 est à Ø
BVC                     \ branche si le bit 6 est à Ø
BVS                     \ branche si le bit 6 est à 1
```

Chapitre 15

La multiplication et la division

La multiplication

Si vous avez bien assimilé tout ce qui précède, vous n'aurez pas de problème pour effectuer une multiplication en code machine Malheureusement le 65Ø2 de l'Oric ne dispose d'aucune instruction de multiplication; il est donc nécessaire d'élaborer un algorithme pour effectuer cette opération. Etudions d'abord la méthode la plus simple pour multiplier entre eux deux petits nombres. Considérons par exemple la multiplication 5 × 6. Nous savons que cela fait 3Ø, mais comment arrivons-nous à ce résultat? Tout simplement en faisant une addition : 5 + 5 + 5 + 5 + 5 + 5 = 3Ø. Il est facile de traduire cela en code machine :

Programme 25

```
 1Ø  REM * * Multiplication simple * *
 2Ø  REM * * HIMEM #96ØØ * *
 3Ø  CODE = #96ØØ
 4Ø  FOR INDEX = Ø TO 19
 5Ø    READ OCTET
 6Ø    POKE CODE + INDEX, OCTET
 7Ø  NEXT INDEX
 8Ø
 9Ø  REM * * Les données du code machine * *
1ØØ  DATA #A9, #ØØ        : REM LDA @#Ø
11Ø  DATA #8D, #ØØ, #Ø4   : REM STA #4ØØ
```

```
120 DATA #A2, #06        : REM LDA @#6
130 DATA #18             : REM CLC
140 DATA #AD, #00, #04   : REM LDA #400
150 DATA #69, #05        : REM ADC @#5
160 DATA #8D, #00, #04   : REM STA #400
170 DATA #CA             : REM DEX
180 DATA #D0, #F5        : REM BNE -11
190 DATA #60             : REM RTS
200
210 REM * * Initialisation et exécution * *
220 CLS
230 CALL (CODE)
240 PRINT PEEK (#400)
```

Tout ce que nous avons fait ici est de mettre en place une boucle qui ajoutera 5 au contenu de l'adresse #400, et cela 6 fois, pour obtenir le résultat recherché. Si cette méthode est acceptable pour multiplier de petits nombres, elle apparaît peu efficace pour des nombres plus grands. Arrivé à ce point, il n'est peut-être pas inutile de revoir comment on effectue la multiplication de deux nombres décimaux. Prenons 123 × 150, par exemple. Nous procéderions ainsi (sans calculatrice, s'il vous plaît !) :

```
    123    (Multiplicande)
  × 150    (Multiplicateur)
  -----
    000    (Produit partiel 1)
   615     (Produit partiel 2)
  123      (Produit partiel 3)
  -----
  18450    (Résultat ou produit final)
```

Les deux valeurs initiales sont appelées multiplicande et multiplicateur, et on obtient leur produit en multipliant, à tour de rôle, chaque chiffre du multiplicateur par le multiplicande. On obtient ainsi un produit partiel, qu'on écrit de sorte que le chiffre de poids le plus faible soit situé exactement sous le chiffre du multiplicateur auquel ce produit correspond. Lorsqu'on a tous les produits partiels, on les additionne pour obtenir le résultat, ou produit final. Nous pouvons appliquer cette technique à des nombres binaires, en commençant par des valeurs de 3 bits, 010 × 011.

```
    010    (Multiplicande)
  × 011    (Multiplicateur)
  -----
    010    (Produit partiel 1)
   010     (Produit partiel 2)
  000      (Produit partiel 3)
  -----
  00110    (Résultat)
```

Si l'on ne tient pas compte des 0 devant le nombre, on arrive au résultat correct 110 (2 × 3 = 6). Reprenons maintenant notre exemple décimal de tout à l'heure et travaillons sur sa représentation binaire :

```
         Ø1111Ø11          123 (#7B)
       × 1ØØ1Ø11Ø          15Ø (#96)
         ØØØØØØØØ
        Ø1111Ø11
       Ø1111Ø11
      ØØØØØØØØ
     Ø1111Ø11
    ØØØØØØØØ
   ØØØØØØØØ
  Ø1111Ø11
  1ØØ1ØØØØØØ1ØØ1Ø          1845Ø (#4812)
```

Vous avez sûrement remarqué que pour un Ø dans le multiplicateur, le produit partiel sera une ligne de Ø (quelque chose multiplié par zéro vaut zéro). C'est pourquoi on peut ignorer purement et simplement un zéro dans le multiplicateur, *mais* il faudra penser à décaler le produit partiel suivant d'autant de chiffres qu'on a sauté de zéros, de manière à ce que le chiffre de poids le plus faible de ce produit partiel se trouve toujours situé sous le chiffre du multiplicateur correspondant. Cette technique d'ignorer les zéros et de décaler le produit suivant peut servir pour écrire un programme de multiplication très efficace.

Programme 26

```
 1Ø  REM * * Mult. de nombres sur un octet * *
 2Ø  REM * * Avec un résultat sur 2 octets * *
 3Ø  REM * * Réinitialiser d'abord HIMEM #96ØØ * *
 4Ø  REM * * Avant de faire tourner le programme * *
 5Ø  CODE = #96ØØ
 6Ø  REPEAT
 7Ø    READ OCTET
 8Ø    POKE CODE + INDEX, OCTET
 9Ø    INDEX = INDEX + 1
1ØØ  UNTIL OCTET = #6Ø
11Ø
12Ø  REM * * Les données du code machine * *
13Ø
14Ø  DATA #A2, #Ø8        : REM LDX @8
15Ø  DATA #A9, #ØØ        : REM LDA @Ø
16Ø  DATA #46, #71        : REM LSR #71
17Ø  DATA #9Ø, #Ø3        : REM BCC +3
18Ø  DATA #18             : REM CLC
19Ø  DATA #65, #7Ø        : REM ADC #7Ø
2ØØ  DATA #6A             : REM ROR A
21Ø  DATA #66, #72        : REM ROR #72
```

```
22Ø  DATA #CA            : REM DEX
23Ø  DATA #DØ, #F3       : REM BNE − 12
24Ø  DATA #85, #73       : REM STA #73
25Ø  DATA #6Ø            : REM RTS
26Ø
27Ø  REM * * Initialisation et calcul * *
28Ø  CLS
29Ø  INPUT "Multiplicande :"; MCANDE
3ØØ  POKE #7Ø, MCANDE
31Ø  INPUT "Multiplicateur :"; MTEUR
32Ø  POKE #71, MTEUR
33Ø  CALL (CODE)
34Ø  PRINT "Le résultat est";
35Ø  PRINT DEEK (#72)
```

Ce programme saisit deux nombres d'un seul octet, les multiplie l'un par l'autre et range le résultat, qui peut occuper 16 bits, en page zéro. Contrairement à ce que nous avons vu dans l'exemple de la multiplication binaire, on ne calcule pas tous les produits partiels avant de les additionner, mais on ajoute chaque résultat partiel au précédent, au fur et à mesure qu'on les obtient. Cette méthode est un peu plus rapide puisqu'on arrive au produit final dès que le dernier bit du multiplicateur a été traité.

La division

Lorsque nous divisons un nombre par un autre, nous cherchons en fait combien de fois on peut soustraire le second nombre du premier. Prenons par exemple 125 ÷ 5 :

```
(Dividende)  125     | 5     (Diviseur)
             −1Ø     |----
             ---     | 25    (Quotient)
              25     |
             −25     |
             ---     |
(Reste)        Ø     |
```

Ici, on peut soustraire deux fois 5 de 12. On note donc le chiffre 2 comme partie du quotient, et on pose 1Ø (deux fois 5) ôté de 12, il reste 2. Comme on ne peut soustraire 5 de 2, on abaisse le chiffre suivant du dividende pour avoir 25 dont on peut soustraire 5 fois 5, sans avoir de reste. On écrit donc 5 comme deuxième chiffre du quotient, donnant ainsi le résultat final. Pour diviser des nombres binaires, on procède de la même façon. L'exemple ci-dessus codé en binaire donne :

```
Ø 1 1 1 1 1 Ø 1     | Ø 1 Ø 1
Ø 1 Ø 1             |----------------
    1 Ø 1           | Ø Ø Ø 1 1 Ø Ø 1
    1 Ø 1           |
      Ø 1 Ø 1       |
        1 Ø 1       |
            Ø       |
```

En fait, comme vous pouvez le constater, il est beaucoup plus simple de diviser des nombres binaires que des nombres décimaux. Si le diviseur est plus petit ou égal au dividende, le bit correspondant du quotient sera à 1. Si la soustraction n'est pas possible, on place un Ø dans le quotient, on abaisse le bit suivant du dividende et on recommence. Le programme suivant fait une division avec des nombres d'un seul octet et indique s'il y a un reste ou non.

Programme 27

```
 1Ø  REM * * Division de nombres sur un octet * *
 2Ø  REM * * Avec un résultat sur 2 octets * *
 3Ø  REM * * Réinitialiser d'abord HIMEM #96ØØ * *
 4Ø  REM * * Avant de faire tourner le programme * *
 5Ø  CODE = #96ØØ
 6Ø  REPEAT
 7Ø    READ OCTET
 8Ø    POKE CODE + INDEX, OCTET
 9Ø    INDEX = INDEX + 1
1ØØ  UNTIL OCTET = #6Ø
11Ø
12Ø  REM * * Les données du code machine * *
13Ø
14Ø  DATA #A2, #Ø8        : REM LDX @8
15Ø  DATA #A9, #ØØ        : REM LDA @Ø
16Ø  DATA #Ø6, #71        : REM ASL #71
17Ø  DATA #2A             : REM ROL A
18Ø  DATA #C5, #7Ø        : REM CMP #7Ø
19Ø  DATA #9Ø, #Ø4        : REM BCC +4
2ØØ  DATA #E5, #7Ø        : REM SBC #7Ø
21Ø  DATA #E6, #71        : REM INC #71
22Ø  DATA #CA             : REM DEX
23Ø  DATA #DØ, #F2        : REM BNE -13
24Ø  DATA #85, #72        : REM STA #72
```

```
25Ø  DATA #6Ø              : REM RTS
26Ø
27Ø  REM * * Initialisation et calcul * *
28Ø  CLS
29Ø  INPUT "Dividende :"; DENDE
3ØØ  POKE #7Ø, DENDE
31Ø  INPUT "Diviseur :"; DSEUR
32Ø  POKE #71, DSEUR
33Ø  CALL (CODE)
34Ø  PRINT "Le résultat est :";
35Ø  PRINT PEEK (#71)
36Ø  PRINT "Reste :";
37Ø  PRINT PEEK (#72)
```

Ce programme utilise les instructions de décalage en lignes 16Ø et 17Ø, comme pour un décalage de deux octets, l'accumulateur jouant le rôle de l'octet haut. La retenue que produit ROL A n'a pas d'importance, en fait c'est Ø, et elle sera détruite par la prochaine instruction ASL #71.

Annexes

1 Les codes ASCII

Le code ASCII (abréviation pour «American Standard Code for Information Interchange»), sert à coder sous forme numérique les lettres et un certain nombre de caractères de contrôle sur pratiquement tous les micro-ordinateurs, pour que ces derniers puissent les traiter. Les codes ASCII de Ø à 31 sont appelés «codes de contrôle», puisqu'ils servent à contrôler différentes opérations exécutées par l'Oric. Les nombres 32 à 127 servent à coder les lettres, les chiffres et les signes de ponctuation. Le tableau A1.2 en donne la liste.

Tableau A1.1

Décimal	Hex.	Signification
Ø	Ø	Nul - Ne fais rien!
1	1	CTRL A
2	2	
3	3	Interruption
4	4	Hauteur double oui/non
5	5	
6	6	Clic de touche oui/non
7	7	Bip interne
8	8	Déplacement curseur d'un pas en arrière
9	9	Déplacement curseur d'un pas en avant
1Ø	A	Déplacement curseur d'un pas vers le bas
11	B	Déplacement curseur d'un pas vers le haut
12	C	Effacement de l'écran texte
13	D	Retour chariot (curseur en début de ligne suivante)
14	E	Effacement d'une ligne
15	F	
16	1Ø	Imprimante marche/arrêt
17	11	Curseur allumé/éteint
18	12	
19	13	Ecran allumé/éteint
2Ø	14	
21	15	
22	16	
23	17	
24	18	Annulation de ligne
25	19	
26	1A	
27	1B	
28	1C	
29	1D	Inversion vidéo oui/non
3Ø	1E	
31	1F	

Table A1.2

Décimal	Hex.	ASCII	Décimal	Hex.	ASCII
32	20	espace	81	51	Q
33	21	!	82	52	R
34	22	“	83	53	S
35	23	#	84	54	T
36	24	$	85	55	U
37	25	%	86	56	V
38	26	&	87	57	W
39	27	’	88	58	X
40	28	(	89	59	Y
41	29	)	90	5A	Z
42	2A	*	91	5B	[
43	2B	+	92	5C	\
44	2C	,	93	5D	]
45	2D	−	94	5E	^
46	2E	.	95	5F	£
47	2F	/	96	60	©
48	30	0	97	61	a
49	31	1	98	62	b
50	32	2	99	63	c
51	33	3	100	64	d
52	34	4	101	65	e
53	35	5	102	66	f
54	36	6	103	67	g
55	37	7	104	68	h
56	38	8	105	69	i
57	39	9	106	6A	j
58	3A	:	107	6B	k
59	3B	;	108	6C	l
60	3C	<	109	6D	m
61	3D	=	110	6E	n
62	3E	>	111	6F	o
63	3F	?	112	70	p
64	40	@	113	71	q
65	41	A	114	72	r
66	42	B	115	73	s
67	43	C	116	74	t
68	44	D	117	75	u
69	45	E	118	76	v
70	46	F	119	77	w
71	47	G	120	78	x
72	48	H	121	79	y
73	49	I	122	7A	z
74	4A	J	123	7B	(
75	4B	K	124	7C	¦
76	4C	L	125	7D	)
77	4D	M	126	7E	~
78	4E	N	127	7F	DEL (effacement)
79	4F	O	128	80	
80	50	P	129	81	

2 Le 65Ø2

Jusqu'ici nous n'avons parlé du 65Ø2 de l'Oric que sur le plan logiciel, en d'autres termes nous avons appris à le programmer. Il serait bon, avant de terminer, de donner un aperçu de ses caractéristiques matérielles. On peut se demander par exemple, quelle est son organisation interne et comment se fait la circulation des données. Bien que ce ne soit pas absolument indispensable, la compréhension de son fonctionnement ne peut qu'enrichir vos connaissances.

La figure A2.1 est un schéma simplifié de l'*architecture* du 65Ø2. En le regardant de près, vous reconnaîtrez facilement un certain nombre d'éléments. D'autres sont nouveaux. Par exemple, les trois bus : le bus d'adresse, le bus de données et le bus de contrôle. Vous vous demandez sans doute ce qu'est un bus. Ce n'est pas, comme vous pourriez le croire, le 38 qui va au Châtelet, c'est simplement un terme générique pour désigner un ensemble de fils ou de lignes d'un circuit imprimé où transitent électroniquement des Ø et des 1.

En mettant une série de Ø et de 1 sur les huits lignes du bus de donnée, on fait circuler un octet d'information entre le microprocesseur et l'emplacement mémoire dont l'adresse est définie au même moment par les seize lignes du bus d'adresse. Les lignes du bus de contrôle véhiculent les nombreux signaux de synchronisation nécessaires au fonctionnement de l'Oric.

L'exécution d'une instruction

Examinons maintenant comment le 65Ø2 recherche, reconnaît et exécute une instruction. Il doit d'abord localiser et lire l'instruction suivante à exécuter dans le programme en code machine. Pour cela, il place le contenu courant du compteur de programme sur le bus d'adresse et un signal de lecture sur la ligne appropriée du bus de contrôle. L'instruction, ou plus exactement l'octet représentant cette instruction, est quasi instantanément placé sur le bus de donnée. Le 65Ø2 transfère alors cet octet dans un registre interne de huit bits, appelé le Registre d'Instruction (RI en abrégé), qui sert exclusivement à stocker une donnée en attente de traitement. A ce moment là, l'Unité de Contrôle interprète l'instruction et génère les signaux internes et externes nécessaires à son exécution. S'il s'agit, par exemple, de l'octet #A5, le 65Ø2 l'interprète comme étant LDA page zéro et cherche un autre octet qu'il considère comme étant l'adresse à laquelle se trouve la donnée à charger dans l'accumulateur. Chacune de ces opérations est effectuée selon le processus qui a été décrit plus haut.

Il est donc très important que le microprocesseur aille chercher les instructions et les données dans le bon ordre. Pour cela, le compteur de programme comprend un dispositif d'incrémentation automatique. Chaque fois que son contenu est transféré sur le bus d'adresse, sa valeur est incrémentée de un, ce qui garantit que les octets seront lus et rangés dans le bon ordre.

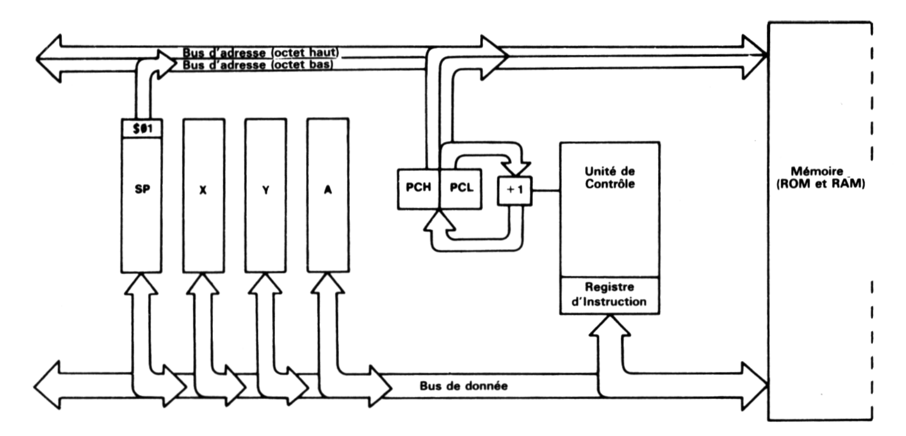


Figure A2.1 Schéma de l'architecture du 65Ø2.

3 Le Jeu d'Instructions

Cette annexe contient la description complète de chacune des 56 instructions du 65Ø2. Pour simplifier la consultation, les mnémoniques sont rangés par ordre alphabétique et la description comporte toujours les subdivisions suivantes :

Définition : Brève description de la fonction de l'instruction.

Tableau : Présentation des modes d'adressages disponibles pour l'instruction, des codes opérateurs correspondants, du nombre d'octets utilisés et des temps d'exécution en cycles pour chaque mode d'adressage.

Registre d'Etat : Configuration du registre d'état après l'exécution de l'instruction. Les conventions suivantes seront utilisées :

* L'indicateur est modifié par l'instruction et son état dépend du résultat de l'opération.
1 L'indicateur est mis à un par l'instruction.
Ø L'indicateur est mis à zéro par l'instruction.

S'il n'y a aucune indication, l'indicateur n'est pas affecté par l'instruction.

Opération : Brève description de l'opération effectuée par l'instruction et des effets sur le registre d'état.

Emploi : Suggestions et conseils sur l'utilisation de cette instruction.

ADC

Addition avec retenue d'un octet de la mémoire à l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
ADC @immédiat	#69	2	2
ADC page zéro	#65	2	3
ADC page zéro, X	#75	2	4
ADC absolu	#6D	3	4
ADC absolu, X	#7D	4	4/5
ADC absolu, Y	#79	3	4/5
ADC (page zéro, X)	#61	2	6
ADC (page zéro), Y	#71	2	5/6

N	V	—	B	D	I	Z	C
*	*					*	*

Opération : Ajoute l'octet spécifié au contenu courant de l'accumulateur. Si l'indicateur de retenue est à 1, il est ajouté au résultat qui se trouve dans l'accumulateur. Si ce résultat est plus grand que #FF (255), l'indicateur de retenue est mis à 1. Si le résultat est nul, l'indicateur Zéro est mis à 1. Le bit 7 de l'accumulateur est copié dans le registre d'état. S'il y a eu débordement du bit 6 sur le bit 7, l'indicateur de débordement est mis à 1.

Emploi : Permet d'additionner des nombres sur un ou plusieurs octets. Les retenues d'un octet sur l'autre sont prises en charge par l'indicateur de retenue qui est inclus dans l'addition.

AND

ET logique entre un octet de la mémoire et l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
AND @immédiat	#29	2	2
AND page zéro	#25	2	3
AND page zéro, X	#35	2	4
AND absolu	#2D	3	4
AND absolu, X	#3D	3	4/5
AND absolu, Y	#39	3	4/5
AND (page zéro, X)	#21	2	6
AND (page zéro), Y	#31	2	5

N	V	—	B	D	I	Z	C
*						*	

Opération : Effectue un ET logique entre les bits de l'octet spécifié et ceux de l'accumulateur. Le résultat de l'opération est dans l'accumulateur et l'octet en mémoire n'est pas modifié. Si le résultat de l'opération est nul, l'indicateur Zéro est mis à 1. Si le bit 7 est à 1, l'indicateur Négatif est mis à 1. Sinon ces deux indicateurs sont mis à zéro.

Emploi : Cette instruction est utilisée pour mettre à zéro certains bits de l'accumulateur.

```
AND @#FØ      \ Mise à zéro du quartet bas, 1111ØØØØ
AND @#ØF      \ Mise à zéro du quartet haut, ØØØØ1111
```

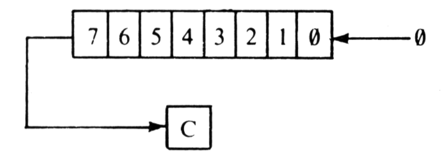
ASL

Décalage d'un bit vers la gauche d'un octet de la mémoire ou de l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
ASL accumulateur	#ØA	1	2
ASL page zéro	#Ø6	2	5
ASL page zéro, X	#16	2	6
ASL absolu	#ØE	3	6
ASL absolu, X	#1E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Opération : Décale tous les bits de l'octet spécifié d'un bit vers la gauche. Le bit 7 passe dans l'indicateur de retenue et le bit Ø est mis à Ø.



L'indicateur de retenue est mis à 1 ou à Ø selon que le bit 7 contenait 1 ou Ø avant le décalage. L'indicateur Négatif est mis à 1 si le bit 6 contenait 1 avant le décalage. L'indicateur Zéro est mis à 1, si l'octet vaut #ØØ après le décalage (c'est à dire qu'il contenait #ØØ ou #8Ø avant le décalage).

Emploi : A pour effet de multiplier un octet par deux. Peut servir à faire passer le quartet bas d'un octet dans le quartet haut.

BCC

Branchement si l'indicateur de retenue est à Ø (C = Ø)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BCC relatif	#9Ø	2	2/3/4

N V — B D I Z C

Opération : Si l'indicateur de retenue est à Ø (C = Ø), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur de retenue est à un (C = 1) le branchement ne s'effectue pas et le 65Ø2 ignore l'octet suivant l'instruction.

Emploi : L'indicateur de retenue est conditionné par plusieurs instructions telles que ADC, SBC, CMP, CPX et CPY et le branchement s'effectue si l'une de ces opérations a pour effet de mettre l'indicateur de retenue à Ø. On peut également «forcer» le branchement en écrivant :

```
CLC                                  \ C = Ø
BCC valeur                           \ branchement
```

BCS

Branchement si l'indicateur de retenue est à 1 (C = 1)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BCS relatif	#BØ	2	2/3/4

N V — B D I Z C

Opération : Si l'indicateur de retenue est à 1 (C = 1), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur de retenue est à Ø (C = Ø) le branchement ne s'effectue pas et le 65Ø2 ignore l'octet suivant l'instruction.

Emploi : Comme pour BCC, le branchement ne s'effectue que si une opération a pour effet de mettre l'indicateur de retenue à 1. On peut également «forcer» le branchement en écrivant :

```
SEC                                  \ C = 1
BCS valeur                           \ branchement
```

BEQ

Branchement si l'indicateur Zéro est à 1 (Z = 1)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BEQ relatif	#FØ	2	2/3/4

N	V	—	B	D	I	Z	C

Opération : Si l'indicateur Zéro est à 1 (Z = 1), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur Zéro est à Ø (Z = Ø) le branchement ne s'effectue pas et le 65Ø2 ignore l'octet suivant l'instruction.

Emploi : Est utilisé pour provoquer un branchement lorsque l'indicateur Zéro est à 1. C'est le cas lorsqu'une opération a pour résultat Ø (ex : LDA @ Ø). BEQ est souvent utilisée après une instruction de comparaison :

CMP @ "?"
BEQ Question
Si la comparaison est vraie, l'indicateur Zéro est mis à 1 et le branchement s'effectue.

BIT

Teste les bits d'un octet en mémoire

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BIT page zéro	#24	2	3
BIT absolu	#2C	3	4

N	V	—	B	D	I	Z	C
*	*					*	

Opération : L'opération BIT ne modifie que le registre d'état. L'accumulateur et l'octet spécifié restent inchangés. Les bits 7 et 6 de l'octet testé sont copiés respectivement dans N et V. L'indicateur Zéro est affecté par un ET logique bit à bit entre l'accumulateur et l'octet en mémoire. Si le résultat du ET est Ø, Z prend la valeur 1, sinon Z = Ø.

Emploi : Est souvent utilisé avec BPL/BMI ou BVS/BVC pour tester les bits 7 et 6 d'un octet en mémoire et effectuer un branchement en fonction de leur état.

BMI

Branchement si l'indicateur Négatif est à 1 (N = 1)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BMI relatif	#3∅	2	2/3/4

N V — B D I Z C

Opération : Si l'indicateur Négatif est à 1 (N = 1), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur Négatif est à ∅ (N = ∅) le branchement ne s'effectue pas et le 65∅2 ignore l'octet suivant l'instruction.
Emploi : En général, après une opération (par exemple LDA, LDX, etc.), le bit 7 du registre est copié dans l'indicateur Négatif. S'il est à un, le branchement BMI s'effectue. Les lettres MI (pour MoIns) du mnémonique soulignent l'importance de cette instruction lorsqu'on utilise l'arithmétique signée, où le bit 7 sert à marquer le signe d'un nombre en complément à deux.

BNE

Branchement si l'indicateur Zéro est à ∅ (Z = ∅)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BNE relatif	#D∅	2	2/3/4

N V — B D I Z C

Opération : Si l'indicateur Zéro est à ∅ (Z = ∅), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur Zéro est à 1 (Z = 1) le branchement ne s'effectue pas et le 65∅2 ignore l'octet suivant l'instruction.

Emploi : Est utilisé pour effectuer un branchement si l'indicateur Zéro est à ∅. Cette instruction est souvent utilisée avec un compteur décroissant, comme test de fin de boucle.

```
DEX
BNE ENCORE
```

qui provoque un branchement arrière tant que X est différent de ∅ et que Z=∅.

BPL

Branchement si l'indicateur Négatif est à Ø (N = Ø)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BPL relatif	#1Ø	2	3/4/5

N	V	—	B	D	I	Z	C

Opération : Si l'indicateur Négatif est à Ø (N = Ø), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur Négatif est à 1 (N = 1) le branchement ne s'effectue pas et le 65Ø2 ignore l'octet suivant l'instruction.

Emploi : En général, après une opération (par exemple LDA, ROL, CPX, etc.), le bit 7 du registre est copié dans l'indicateur Négatif. S'il est à zéro, le branchement BPL s'effectue. Les lettres PL (pour PLus) du mnémonique soulignent l'importance de cette instruction lorsqu'on utilise l'arithmétique signée, où le bit 7 sert à marquer le signe d'un nombre en complément à deux. Si on utilise un compteur décroissant dans une boucle, cette instruction permet d'exécuter la boucle tant que le compteur est supérieur ou égal à zéro.

```
DEX
BPL ENCORE
```

Cette boucle se terminera quand X passera de Ø à #FF, puisque #FF = 11111111 en binaire et que le bit 7 est donc à 1.

BRK

Force une interruption logicielle.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BRK inhérent	#ØØ	1	7

N	V	—	B	D	I	Z	C
			1				

Opération : L'adresse contenue dans le compteur de programme plus 1, puis le registre d'état sont sauvegardés sur la pile.

Emploi : Est utilisée pour provoquer une interruption logicielle.

BVC

Branchement si l'indicateur de débordement est à ∅ (V = ∅)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BVC relatif	#5∅	2	2/3/4

N V — B D I Z C

Opération : Si l'indicateur de débordement est à ∅ (V= ∅), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur de débordement est à 1 (V = 1) le branchement ne s'effectue pas et le 65∅2 ignore l'octet suivant l'instruction.

Emploi : Est utilisé pour détecter un débordement (c'est-à-dire une retenue) du bit 6 sur le bit 7, lorsqu'on utilise l'arithmétique signée. L'addition de deux nombres de signe opposé *ne peut pas* provoquer un débordement, mais une addition de deux nombres de même signe *peut* le faire. Par exemple :

```
  ∅ 1 ∅ ∅ 1 1 1 1      (#4F)
+ ∅ 1 ∅ ∅ ∅ ∅ ∅ ∅      (#4∅)
  ---------------
  1 ∅ ∅ ∅ 1 1 1 1      (– #71)
  ↑
  └──── Débordement du bit 6 sur le bit 7
```

Le résultat est négatif, ce qui est absurde ! De même l'addition de deux grands nombres négatifs peut produire un résultat positif. En fait, le débordement ne peut se produire que dans les cas suivants :

1. Addition de deux grands nombres positifs.
2. Addition de deux grands nombres négatifs.
3. Soustraction d'un grand nombre négatif d'un grand nombre positif.
4. Soustraction d'un grand nombre positif d'un grand nombre négatif.

L'indicateur de débordement est utilisé pour signaler le débordement du bit 6 sur le bit 7, donc un changement de signe en arithmétique signée. S'il est à zéro, il n'y a pas eu de débordement et le branchement BVC a lieu. On peut « forcer » ce branchement en écrivant :

CLV	\ met l'indicateur de débordement à zéro
BVC forcé	\ branche

BVS

Branchement si l'indicateur de débordement est à 1 (V = 1)

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
BVS relatif	#7Ø	2	2/3/4

N	V	—	B	D	I	Z	C

Opération : Si l'indicateur de débordement est à 1 (V = 1), l'octet qui suit l'instruction est interprété comme un nombre en complément à 2 et est ajouté au contenu courant du compteur de programme. On obtient ainsi l'adresse à partir de laquelle l'exécution du programme doit être poursuivie. On peut faire un branchement arrière de 126 octets ou un branchement avant de 129 octets. Si l'indicateur de débordement est à Ø (V = Ø) le branchement ne s'effectue pas et le 65Ø2 ignore l'octet suivant l'instruction.

Emploi : Est utilisé pour provoquer un branchement si le signe d'un nombre a été changé. BVS est employé la plupart du temps lorsqu'on utilise l'arithmétique signée. Voir BVC pour plus de détails.

CLC

Mise à Ø de l'indicateur de retenue (C = Ø).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CLC inhérent	#18	1	2

N	V	—	B	D	I	Z	C
							Ø

Opération : Met l'indicateur de retenue à Ø.

Emploi : Doit toujours être utilisé avant une addition, car l'indicateur de retenue est toujours ajouté par ADC. On peut «forcer» un branchement par :

```
CLC                     \ mise à Ø de l'indicateur de retenue
BCC forcé               \ et «saut»
```

CLD

Mise à ∅ de l'indicateur Décimal (D = ∅).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CLD inhérent	#D8	1	2

N	V	—	B	D	I	Z	C
				∅			

Opération : Met l'indicateur Décimal à ∅.

Emploi : Est utilisé pour remettre le 65∅2 en mode hexadécimal, qui est le mode normal, par opposition au mode décimal.

CLI

Mise à ∅ de l'indicateur d'Interruption IRQ (I = ∅).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CLI inhérent	#58	1	2

N	V	—	B	D	I	Z	C
					∅		

Opération : Met l'indicateur d'Interrution IRQ à ∅.

Emploi : Autorise le traitement des interruptions IRQ après exécution de l'instruction en cours.

CLV

Mise à Ø de l'indicateur de débordement (V = Ø).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CLV inhérent	#B8	1	2

N	V	—	B	D	I	Z	C
	Ø						

Opération : Met l'indicateur de débordement à Ø.

Emploi : Utilisé pour mettre à Ø l'indicateur de débordement après un débordement du bit 6 sur le bit 7. La plupart du temps, cela n'a d'importance que si l'on utilise l'arithmétique signée.

CMP

Compare un octet de la mémoire à l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CMP @immédiat	#C9	2	2
CMP page zéro	#C5	2	3
CMP page zéro, X	#D5	2	4
CMP absolu	#CD	3	4
CMP absolu, X	#DD	3	4/5
CMP absolu, Y	#D9	3	4/5
CMP (page zéro, X)	#C1	2	6
CMP (page zéro), Y	#D1	2	5/6



Opération : L'octet en mémoire spécifié (ou la valeur immédiate) est soustraite du contenu de l'accumulateur. Les contenus de l'octet en mémoire et de l'accumulateur NE SONT PAS modifiés, mais les indicateurs Négatif, Zéro et de retenue sont modifiés en fonction du résultat de la soustraction. Pour faire cette soustraction, le 65Ø2 commence par mettre à 1 l'indicateur de retenue, puis ajoute au contenu de l'accumulateur le complément à 2 de l'octet de la mémoire. Si les deux valeurs sont égales (mémoire = accumulateur) l'indicateur Zéro est mis à 1 et l'indicateur de retenue reste à 1. Si le contenu de l'octet est plus petit que le contenu de l'accumulateur (mémoire < accumulateur), l'indicateur Zéro est mis à Ø et l'indicateur de retenue est mis à 1. Si la valeur de l'octet est supérieure à celle de l'accumulateur (mémoire > accumulateur), les deux indicateurs sont mis à Ø. Si on utilise du binaire non signé, l'indicateur Négatif est aussi mis à 1. Quand on utilise du binaire signé, il faut tester l'indicateur de débordement en plus de l'indicateur Négatif, pour vérifier si l'on a un «vrai» résultat négatif.

Emploi : Permet de faire des tests intermédiaires quand on ne peut pas tester directement le registre d'état. Par exemple :

```
CMP @#ØØ
BEQ AILLEURS
```

fait perdre deux octets pour rien, puisque l'indicateur Zéro est 1 si l'accumulateur contient #ØØ. Il suffit donc d'employer BEQ AILLEURS.

CPX

Compare un octet de la mémoire au registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CPX @immédiat	#EØ	2	2
CPX page zéro	#E4	2	3
CPX absolu	#EC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Opération : L'octet en mémoire spécifié (ou la valeur immédiate) est soustraite du contenu du registre X. Les contenus de l'octet en mémoire et du registre X NE SONT PAS modifiés, mais les indicateurs Négatif, Zéro et de retenue sont modifiés en fonction du résultat de la soustraction. Pour faire cette soustraction, le 65Ø2 commence par mettre à 1 l'indicateur de retenue, puis ajoute au contenu du registre X le complément à 2 de l'octet de la mémoire. Si les deux valeurs sont égales (mémoire = registre X) l'indicateur Zéro est mis à 1 et l'indicateur de retenue reste à 1. Si le contenu de l'octet est plus petit que le contenu du registre X (mémoire < registre X), l'indicateur Zéro est mis à Ø et l'indicateur de retenue reste à 1. Si la valeur de l'octet est supérieure à celle du registre X (mémoire > registre X), les deux indicateurs sont mis à Ø. Si on utilise du binaire non signé, l'indicateur Négatif est aussi mis à 1.

Emploi : Permet de faire des tests intermédiaires quand on ne peut pas tester directement le registre d'état. Par exemple, pour tester le contenu du registre X lorsqu'il sert de compteur dans une boucle, on peut faire :

```
        LDX @#EØ       \ charge #EØ dans X
ENCORE  DEX            \ décrémente X
        CPX @#87       \ est-ce que X = #87?
        BNE ENCORE     \ non, on recommence
```

CPY

Compare un octet de la mémoire au registre Y.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
CPY @immédiat	#CØ	2	2
CPY page zéro	#C4	2	3
CPY absolu	#CC	3	4

N	V	—	B	D	I	Z	C
*						*	*

Opération : L'octet en mémoire spécifié (ou la valeur immédiate) est soustraite du contenu du registre Y. Les contenus de l'octet en mémoire et du registre Y NE SONT PAS modifiés, mais les indicateurs Négatif, Zéro et de retenue sont modifiés en fonction du résultat de la soustraction. Pour faire cette soustraction, le 65Ø2 commence par mettre à 1 l'indicateur de retenue, puis ajoute au contenu du registre Y le complément à 2 de l'octet de la mémoire. Si les deux valeurs sont égales (mémoire = registre Y) l'indicateur Zéro est mis à 1 et l'indicateur de retenue reste à 1. Si le contenu de l'octet est plus petit que le contenu du registre Y (mémoire < registre Y), l'indicateur Zéro est mis à Ø et l'indicateur de retenue reste à 1. Si la valeur de l'octet est supérieure à celle du registre Y (mémoire > registre Y), les deux indicateurs sont mis à Ø. Si on utilise du binaire non signé, l'indicateur Négatif est aussi mis à 1.

Emploi : Permet de faire des tests intermédiaires quand on ne peut pas tester directement le registre d'état. Par exemple, pour tester le contenu du registre Y lorsqu'il sert de compteur dans une boucle, on peut faire :

```
        LDY @#EØ         \ charge #EØ dans Y
ENCORE  DEY              \ décrémente Y
        CPY @#87         \ est-ce que Y = #87?
        BNE ENCORE       \ non, on recommence
```

DEC

Décrémentation d'un octet en mémoire.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
DEC page zéro	#C6	2	5
DEC page zéro, X	#D6	2	6
DEC absolu	#CE	3	6
DEC absolu, X	#DE	3	7

N	V	—	B	D	I	Z	C
*						*	

Opération : L'octet à l'adresse spécifiée est décrémenté de 1 (MEMOIRE = MEMOIRE − 1). Si le résultat de l'opération est Ø, l'indicateur Zéro est mis à 1. Le bit 7 de l'octet est copié dans l'indicateur Négatif.

Emploi : Est utilisé pour retrancher 1 d'un compteur rangé en mémoire.

DEX

Décrémentation du contenu du registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
DEX inhérent	#CA	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : On retranche 1 de la valeur courante contenue dans le registre X (X = X − 1). Si le résultat de l'opération est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif (N = Ø si X < #8Ø; N = 1 si X > #7F). L'indicateur de retenue n'est pas modifié par cette instruction.

Emploi : Est utilisé quand le registre X sert de déplacement par rapport à une adresse de base dans l'adressage indexé, ce qui permet d'accéder à une suite d'octets. Est systématiquement utilisé pour décrémenter le registre X quand il sert de compteur dans une boucle qui comprend une instruction de branchement jusqu'à ce que X = Ø (Z = 1).

DEY

Décrémentation du contenu du registre Y.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
DEY inhérent	#88	1	2

N	V	—	B	D	Z	C
*					*	

Opération : On retranche 1 de la valeur courante contenue dans le registre Y (Y = Y − 1). Si le résultat de l'opération est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif (N = Ø si Y < #8Ø; N = 1 si Y > #7F). L'indicateur de retenue n'est pas modifié par cette instruction.

Emploi : Est utilisé quand le registre Y sert de déplacement par rapport à une adresse de base dans l'adressage indexé, ce qui permet d'accéder à une suite d'octets. Est systématiquement utilisé pour décrémenter le registre Y quand il sert de compteur dans une boucle qui comprend une instruction de branchement jusqu'à ce que Y = Ø (Z = 1).

EOR

OU exclusif entre un octet de la mémoire et l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
EOR @immédiat	#49	2	2
EOR page zéro	#45	2	3
EOR page zéro, X	#55	2	4/5
EOR absolu	#4D	3	4
EOR absolu, X	#5D	3	4/5
EOR absolu, Y	#59	3	4/5
EOR (page zéro, X)	#41	2	6
EOR (page zéro), Y	#51	2	5/6

N	V	—	B	D	Z	C
*					*	

Opération : Effectue un OU exclusif bit à bit entre l'octet spécifié et le contenu courant de l'accumulateur. Si le résultat, qui est dans l'accumulateur, est nul, l'indicateur Zéro est mis à 1. Le bit 7 de l'accumulateur est copié dans l'indicateur Négatif.

Emploi : Est utilisé pour obtenir le complément ou l'inverse d'un octet de donnée.

INC

Incrémentation d'un octet en mémoire.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
INC page zéro	#E6	2	5
INC page zéro, X	#F6	2	6
INC absolu	#EE	3	6
INC absolu, X	#FE	3	7

N	V	—	B	D	I	Z	C
*						*	

Opération : L'octet à l'adresse spécifiée est incrémenté de 1. Si l'adresse contient ∅ après l'opération, l'indicateur Zéro est mis à 1. Le bit 7 de l'octet est copié dans l'indicateur Négatif.

Emploi : Est utilisé pour ajouter 1 à un compteur rangé en mémoire.

INX

Incrémentation du contenu du registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
INX inhérent	#E8	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : On ajoute 1 à la valeur courante contenue dans le registre X (X = X + 1). Si le résultat de l'opération est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif (N = Ø si X < #8Ø; N = 1 si X > #7F). L'indicateur de retenue n'est pas modifié par cette instruction.

Emploi : Est utilisé quand le registre X sert de déplacement par rapport à une adresse de base dans l'adressage indexé, ce qui permet d'accéder à une suite d'octets. Souvent utilisé pour incrémenter le registre X quand il sert à compter le nombre de passages dans une boucle.

INY

Incrémentation du contenu du registre Y.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
INY inhérent	#C8	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : On ajoute 1 à la valeur courante contenue dans le registre Y (Y = Y + 1). Si le résultat de l'opération est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif. L'indicateur de retenue n'est pas modifié par cette instruction.

Emploi : Est utilisé quand le registre Y sert de déplacement par rapport à une adresse de base dans l'adressage indexé, ce qui permet d'accéder à une suite d'octets. Souvent utilisé pour incrémenter le registre Y quand il sert à compter le nombre de passages dans une boucle.

JMP

Saut à une adresse spécifiée.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
JMP absolu	#4C	3	3
JMP (indirect)	#6C	3	3

N	V	—	B	D	I	Z	C

Opération : Pour JMP absolu, les deux octets qui suivent l'instruction sont placés dans le compteur de programme. Dans le cas d'un saut indirect, les deux octets qui se trouvent à l'adresse spécifiée après l'instruction sont chargés dans le compteur de programme.

Emploi : Transfère, de manière non conditionnelle, le contrôle à une autre partie d'un programme, rangée ailleurs dans la mémoire.

JSR

Saut avec sauvegarde de l'adresse de retour.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
JSR absolu	#2Ø	3	6

N	V	—	B	D	I	Z	C

Opération : Agit comme un appel à un sous-programme, en transférant le contrôle à une autre partie de la mémoire jusqu'à une instruction RTS. Le contenu courant, plus 2, du compteur de programme est sauvegardé sur la pile. Le pointeur de pile est incrémenté deux fois. L'adresse absolue qui suit JSR est alors placée dans le compteur de programme et l'exécution se poursuivra à partir de cette nouvelle adresse.

Emploi : Permet de n'écrire qu'une fois une longue séquence de codes, dont on veut se servir plusieurs fois à l'intérieur d'un même programme, puis de l'appeler comme sous-programme, aussi souvent que nécessaire

LDA

Charge l'octet spécifié dans l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
LDA @immédiat	#A9	2	2
LDA page zéro	#A5	2	3
LDA page zéro, X	#B5	2	4
LDA absolu	#AD	3	4
LDA absolu, X	#BD	3	4/5
LDA absolu, Y	#B9	3	4/5
LDA (page zéro, X)	#A1	2	6
LDA (page zéro), Y	#B1	2	5/6

N	V	—	B	D	I	Z	C
*						*	

Opération : Charge la valeur qui suit immédiatement l'instruction, ou le contenu de l'adresse spécifiée après l'instruction, dans l'accumulateur. Si la valeur chargée est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : C'est sans doute l'instruction la plus utilisée, elle permet de manipuler des données, et facilite les opérations arithmétiques et logiques.

LDX

Charge l'octet spécifié dans le registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
LDX @immédiat	#A2	2	2
LDX page zéro	#A6	2	3
LDX page zéro, Y	#B6	2	4
LDX absolu	#AE	3	4
LDX absolu, Y	#BE	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Opération : Charge la valeur qui suit immédiatement l'instruction, ou le contenu de l'adresse spécifiée après l'instruction, dans le registre X. Si la valeur chargée est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Sert à transférer des données pour les traiter ou les ranger. Permet aussi d'initialiser un compteur de boucle.

LDY

Charge l'octet spécifié dans le registre Y.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
LDY @immédiat	#AØ	2	2
LDY page zéro	#A4	2	3
LDY page zéro, X	#B4	2	4
LDY absolu	#AC	3	4
LDY absolu, X	#BC	3	4/5

N	V	—	B	D	I	Z	C
*						*	

Opération : Charge la valeur qui suit immédiatement l'instruction, ou le contenu de l'adresse spécifiée après l'instruction, dans le registre Y. Si la valeur chargée est Ø, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Sert à transférer des données pour les traiter ou les ranger. Permet aussi d'initialiser un compteur de boucle.

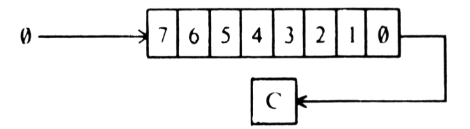
LSR

Décale l'octet spécifié d'un bit vers la droite.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
LSR accumulateur	#4A	1	2
LSR page zéro	#46	2	5
LSR page zéro, X	#56	2	6
LSR absolu	#4E	3	6
LSR absolu, X	#5E	3	7

N	V	—	B	D	I	Z	C
Ø						*	*

Opération : Décale le contenu de l'octet spécifié d'un bit vers la droite, en mettant un Ø dans le bit 7 et le bit Ø dans l'indicateur de retenue.



L'indicateur Négatif est mis à Ø, et l'indicateur de retenue est conditionné par le bit Ø de l'octet décalé. L'indicateur Zéro est mis à 1 si l'octet contient Ø après le décalage (ce qui signifie qu'il contenait #ØØ ou #Ø1 avant l'opération).

Emploi : Divise la valeur d'un octet par deux (si D = Ø) et met le reste dans l'indicateur de retenue. Sert également à faire passer le quartet haut en position de quartet bas.

NOP

Aucune opération.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
NOP inhérent	#EA	1	2

N	V	—	B	D	I	Z	C

Opération : Ne fait rien, sinon incrémenter le compteur de programme.

Emploi : Sert à créer un délai de 2 cycles.

ORA

OU logique entre un octet de la mémoire et l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
ORA @immédiat	#Ø9	2	2
ORA page zéro	#Ø5	2	3
ORA page zéro, X	#15	2	4
ORA absolu	#ØD	3	4
ORA absolu, X	#1D	3	4/5
ORA absolu, Y	#19	3	4/5
ORA (page zéro, X)	#Ø1	2	6
ORA (page zéro), Y	#11	2	5

N	V	—	B	D	I	Z	C
*						*	

Opération : Effectue un OU logique bit à bit entre le contenu de l'accumulateur et la valeur spécifiée ou un octet en mémoire. Le résultat se trouve dans l'accumulateur. Si le bit 7 du résultat est à 1, l'indicateur Négatif est à 1, sinon il est à Ø.

Emploi : Permet de «forcer» certains bits à 1. Par exemple :

ORA @#8Ø \ 1ØØØØØØØ en binaire

permet de forcer le bit 7 à contenir 1.

PHA

Empile le contenu de l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
PHA inhérent	#48	1	3

N	V	—	B	D	I	Z	C

Opération : Le contenu de l'accumulateur est copié à l'emplacement indiqué par le pointeur de pile, qui est ensuite décrémenté de 1.

Emploi : Permet de sauvegarder temporairement des octets en mémoire. On peut sauvegarder également les contenus des registres d'index en les tranférant d'abord dans l'accumulateur. Pour sauvegarder des octets en mémoire, il faut au préalable les charger dans l'accumulateur. On pourra les récupérer avec PLA.

PHP

Empile le contenu du registre d'état.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
PHP inhérent	#Ø8	1	3

N	V	—	B	D	I	Z	C

Opération : Le contenu du registre d'état est copié à l'emplacement indiqué par le pointeur de pile, qui est ensuite décrémenté de 1.

Emploi : Permet de sauvegarder les indicateurs, par exemple avant l'appel d'un sous-programme. On pourra les retrouver avec PLP.

PLA

Charge le sommet de la pile dans l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
PLA inhérent	#68	1	4

N	V	—	B	D	I	Z	C
*						*	

Opération : Le pointeur de pile est incrémenté de 1, et l'octet sur lequel il pointe est copié dans l'accumulateur. Si cet octet vaut #ØØ, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Effectue l'opération inverse de PHA pour dépiler des données préalablement empilées.

PLP

Charge le sommet de la pile dans le registre d'état.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
PLP inhérent	#28	1	4

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Opération : Le pointeur de pile est incrémenté de 1, et le contenu de l'octet sur lequel il pointe est copié dans le registre d'état.

Emploi : Effectue l'opération inverse de PHP pour retrouver le contenu du registre d'état préalablement empilé, ou pour positionner certains indicateurs au moyen d'un octet empilé par PHA.

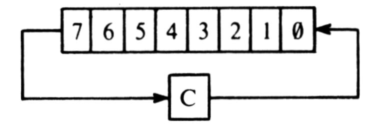
ROL

Rotation d'un octet en mémoire ou de l'accumulateur, d'un bit vers la gauche, par l'intermédiaire de l'indicateur de retenue.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
ROL accumulateur	#2A	1	2
ROL page zéro	#26	2	5
ROL page zéro, X	#36	2	6
ROL absolu	#2E	3	6
ROL absolu, X	#3E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Opération : Effectue une rotation à gauche de l'octet spécifié et du contenu de l'indicateur de retenue.



Le bit 7 passe dans l'indicateur de retenue, qui passe dans le bit Ø. Les autres bits sont décalés à gauche. L'indicateur Négatif est mis à 1, si l'ancien bit 6 était à 1, sinon il est nul. L'indicateur de retenue prend la valeur du bit 7. Si l'octet vaut Ø après la rotation, l'indicateur Zéro est mis à 1.

Emploi : Utilisé avec ASL, il peut servir à doubler la valeur d'un nombre sur plusieurs octets, puisque l'indicateur de retenue permet de transmettre le débordement d'un octet à l'autre. On peut aussi s'en servir pour tester certains bits en les faisant passer dans les indicateurs Négatif, Zéro et de retenue.

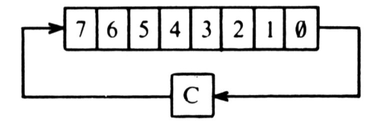
ROR

Rotation d'un octet en mémoire ou de l'accumulateur, d'un bit vers la droite, par l'intermédiaire de l'indicateur de retenue.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
ROR accumulateur	#6A	1	2
ROR page zéro	#66	2	5
ROR page zéro, X	#76	2	6
ROR absolu	#6E	3	6
ROR absolu, X	#7E	3	7

N	V	—	B	D	I	Z	C
*						*	*

Opération : Effectue une rotation à droite de l'octet spécifié et du contenu de l'indicateur de retenue.



Le bit Ø passe dans l'indicateur de retenue, qui passe dans le bit 7. Les autres bits sont décalés à droite. L'indicateur Négatif est mis à 1, si l'indicateur de retenue était à 1, sinon il est nul. L'indicateur de retenue prend la valeur du bit Ø. Si l'octet vaut Ø après la rotation, l'indicateur Zéro est mis à 1.

Emploi : Utilisé avec LSR, il peut servir à diviser par deux la valeur d'un nombre sur plusieurs octets. On peut aussi s'en servir pour tester certains bits en les faisant passer dans les indicateurs Négatif, Zéro et de retenue.

RTI

Retour d'interruption.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
RTI inhérent	#40	1	6

N	V	—	B	D	I	Z	C
*	*		*	*	*	*	*

Opération : Cette instruction s'attend à trouver trois octets sur la pile. Le premier est dépilé et placé dans le registre d'état (tous les indicateurs sont donc affectés). Les deux octets suivants sont placés dans le compteur de programme. Le pointeur de pile est incrémenté chaque fois qu'un octet est dépilé.

Emploi : Est utilisé pour rendre le contrôle à un programme après une interruption. Au moment de l'interruption le microprocesseur avait empilé le compteur de programme et le registre d'état.

RTS

Retour de sous-programme.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
RTS inhérent	#60	1	6

N	V	—	B	D	I	Z	C

Opération : L'adresse placée dans les deux premiers octets de la pile est dépilée, incrémentée de 1 et mise dans le compteur de programme. L'exécution du programme continue à cette adresse. Le pointeur de pile est incrémenté de 2.

Emploi : Rend le contrôle au programme qui a appelé le sous-programme. C'est donc nécessairement la dernière instruction de tout sous-programme.

SBC

Soustraction d'un octet de la mémoire à l'accumulateur avec retenue.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
SBC @immédiat	#E9	2	2
SBC page zéro	#E5	2	3
SBC page zéro, X	#F5	2	4
SBC absolu	#ED	3	4
SBC absolu, X	#FD	3	4/5
SBC absolu, Y	#F9	3	4/5
SBC (page zéro, X)	#E1	2	6
SBC (page zéro), Y	#F1	2	5/6

N	V	—	B	D	I	Z	C
*	*					*	*

Opération : Soustrait la valeur immédiate ou l'octet d'adresse spécifiée du contenu courant de l'accumulateur. Si la valeur à soustraire est plus grande que celle contenue dans l'accumulateur, la retenue sera «empruntée» à l'indicateur de retenue, qui doit donc être mis à 1 avant la soustraction. Si l'indicateur de retenue est à Ø après l'opération, c'est qu'on a eu besoin d'une retenue pour l'effectuer. Si le résultat est nul, l'indicateur Zéro est mis à 1. Le bit 7 de l'accumulateur est copié dans l'indicateur négatif. S'il y a eu débordement du bit 6 sur le bit 7, l'indicateur de débordement est mis à 1.

Emploi : Permet de soustraire des nombres sur un ou plusieurs octets.

SEC

Met l'indicateur de retenue à1 (C = 1).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
SEC inhérent	#38	1	2

N	V	—	B	D	I	Z	C
							1

Opération : Place 1 dans l'indicateur de retenue.

Emploi : Doit toujours être utilisé avant une soustraction, puisque SBC utilise l'indicateur de retenue.

SED

Fait passer en mode décimal en mettant l'indicateur Décimal à 1 (D = 1).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
SED inhérent	#F8	1	2

N	V	—	B	D	I	Z	C
				1			

Opération : Place 1 dans l'indicateur Décimal.

Emploi : Fait passer l'Oric en mode décimal, pour les calculs en Décimal Codé Binaire (DCB). L'indicateur de retenue signalera une retenue sur les centaines, puisque la valeur maximale pouvant être codée sur un octet en DCB est 99.

SEI

Met l'indicateur d'Interruption à 1 (I = 1).

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
SEI inhérent	#78	1	2

N	V	—	B	D	I	Z	C
					1		

Opération : Place 1 dans l'indicateur d'Interruption.

Emploi : Lorsque cet indicateur est à 1 *aucune* interruption IRQ n'est traitée. Mais les interruptions NMI sont prises en compte, ainsi que les BREAK.

STA

Range le contenu de l'accumulateur en mémoire.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
STA page zéro	#85	2	3
STA page zéro, X	#95	2	4
STA absolu	#8D	3	4
STA absolu, X	#9D	3	5
STA absolu, Y	#99	3	5
STA (page zéro, X)	#81	2	6
STA (page zéro), Y	#91	2	6

N V — B D I Z C

Opération : Le contenu de l'accumulateur est copié dans l'octet d'adresse spécifiée.

Emploi : Sert à sauvegarder le contenu de l'accumulateur ou à initialiser des zones de mémoire à une valeur donnée. Permet de transférer des blocs de mémoire d'une zone à une autre lorsqu'on l'utilise avec LDA.

STX

Range le contenu du registre X dans la mémoire.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
STX page zéro	#86	2	3
STX page zéro, Y	#96	2	4
STX absolu	#8E	3	4

N V — B D I Z C

Opération : Le contenu du registre X est copié dans l'octet d'adresse spécifiée.

Emploi : Sert à sauvegarder le contenu du registre X ou à initialiser des zones de mémoire à une valeur donnée.

STY

Range le contenu du registre Y dans la mémoire.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
STY page zéro	#84	2	3
STY page zéro, X	#94	2	4
STY absolu	#8C	3	4

N	V	—	B	D	I	Z	C

Opération : Le contenu du registre Y est copié dans l'octet d'adresse spécifiée.

Emploi : Sert à sauvegarder le contenu du registre Y ou à initialiser des zones de mémoire à une valeur donnée.

TAX

Transfère le contenu de l'accumulateur dans le registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TAX inhérent	#AA	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : Le contenu de l'accumulateur est copié dans le registre X. Si le registre X contient alors zéro, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Sert à sauvegarder temporairement le contenu de l'accumulateur ou à initialiser le registre X lorsqu'il est utilisé comme compteur de boucle. Employé fréquemment après PLA pour retrouver le contenu du registre X précédemment empilé.

TAY

Transfère le contenu de l'accumulateur dans le registre Y.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TAY inhérent	#A8	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : Le contenu de l'accumulateur est copié dans le registre Y. Si le registre Y contient zéro après l'opération, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Sert à sauvegarder temporairement le contenu de l'accumulateur ou à initialiser le registre Y lorsqu'il est utilisé comme compteur de boucle. Employé fréquemment après PLA pour retrouver le contenu du registre Y précédemment empilé.

TSX

Transfère le contenu du pointeur de pile dans le registre X.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TSX inhérent	#BA	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : Le contenu du pointeur de pile est copié dans le registre X. Si le registre X contient zéro après l'opération, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Sert à calculer l'espace disponible sur la pile, ou à sauvegarder le contenu courant du pointeur de pile pendant un test sur les octets empilés.

TXA

Tansfère le contenu du registre X dans l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TXA inhérent	#8A	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : Le contenu du registre X est copié dans l'accumulateur. Si l'accumulateur contient zéro après l'opération, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Permet d'effectuer des opérations logiques ou arithmétiques sur le contenu du registre X. Suivi de PHA, permet de sauvegarder la valeur du registre X sur la pile.

TXS

Transfère le contenu du registre X dans le pointeur de pile.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TXS inhérent	#9A	1	2

N	V	—	B	D	I	Z	C

Opération : Le contenu du registre X est copié dans le pointeur de pile.

Emploi : Permet d'initialiser ou de donner une valeur précise au pointeur de pile. Par exemple :

```
LDX @#FF
TXS
```

pour «vider» la pile et réinitialiser le pointeur de pile.

TYA

Transfère le contenu du registre Y dans l'accumulateur.

Mode d'adressage	Code opérateur	Nb. d'octets	Nb. de cycles
TYA inhérent	#98	1	2

N	V	—	B	D	I	Z	C
*						*	

Opération : Le contenu du registre Y est copié dans l'accumulateur. Si l'accumulateur contient zéro après l'opération, l'indicateur Zéro est mis à 1. Le bit 7 est copié dans l'indicateur Négatif.

Emploi : Permet d'effectuer des opérations logiques ou arithmétiques sur le contenu du registre Y. Suivi de PHA, permet de sauvegarder la valeur du registre Y sur la pile.

4 Tables de Calcul de Branchement

On utilise les tables de calcul de branchement pour trouver la valeur hexadécimale d'un déplacement. Comptez d'abord le nombre d'octets de votre déplacement. Localisez ce nombre à l'intérieur du tableau et vous pourrez lire la valeur hexadécimale du quartet haut dans la colonne de gauche et celle du quartet bas dans la ligne supérieure.

Exemple : Pour un branchement arrière de 16 octets :

Trouvez le nombre 16 dans le tableau A4.1 (dernière ligne); vous avez donc la valeur du quartet haut (#F) et du quartet bas (#Ø). La valeur du déplacement est donc (#FØ).

Tableau A4.1 Table de calcul des branchements arrières

LSD / MSD	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	12Ø	119	118	117	116	115	114	113
9	112	111	11Ø	1Ø9	1Ø8	1Ø7	1Ø6	1Ø5	1Ø4	1Ø3	1Ø2	1Ø1	1ØØ	99	98	97
A	96	95	94	93	92	91	9Ø	89	88	87	86	85	84	83	82	81
B	8Ø	79	78	77	76	75	74	73	72	71	7Ø	69	68	67	66	65
C	64	63	62	61	6Ø	59	58	57	56	55	54	53	52	51	5Ø	49
D	48	47	46	45	44	43	42	41	4Ø	39	38	37	36	35	34	33
E	32	31	3Ø	29	28	27	26	25	24	23	22	21	2Ø	19	18	17
F	16	15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1

Tableau A4.2 Table de calcul des branchements avants

LSD / MSD	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
1	16	17	18	19	2Ø	21	22	23	24	25	26	27	28	29	3Ø	31
2	32	33	34	35	36	37	38	39	4Ø	41	42	43	44	45	46	47
3	48	49	5Ø	51	52	53	54	55	56	57	58	59	6Ø	61	62	63
4	64	65	66	67	68	69	7Ø	71	72	73	74	75	76	77	78	79
5	8Ø	81	82	83	84	85	86	87	88	89	9Ø	91	92	93	94	95
6	96	97	98	99	1ØØ	1Ø1	1Ø2	1Ø3	1Ø4	1Ø5	1Ø6	1Ø7	1Ø8	1Ø9	11Ø	111
7	112	113	114	115	116	117	118	119	12Ø	121	122	123	124	125	126	127

5 Les Codes Opérateurs du 65Ø2

Tous les nombres sont en hexadécimal.

ØØ BRK inhérent
Ø1 ORA (Page zéro, X)
Ø2 Extension future
Ø3 Extension future
Ø4 Extension future
Ø5 ORA Page zéro
Ø6 ASL Page zéro
Ø7 Extension future
Ø8 PHP inhérent
Ø9 ORA #immédiat
ØA ASL accumulateur
ØB Extension future
ØC Extension future
ØD ORA absolu
ØE ASL absolu
ØF Extension future
1Ø BPL relatif
11 ORA (Page zéro), Y
12 Extension future
13 Extension future
14 Extension future
15 ORA Page zéro, X
16 ASL Page zéro, X
17 Extension future
18 CLC inhérent
19 ORA absolu, Y
1A Extension future
1B Extension future
1C Extension future
1D ORA absolu, X
1E ASL absolu, X
1F Extension future
2Ø JSR absolu
21 AND (Page zéro, X)
22 Extension future
23 Extension future
24 BIT Page zéro
25 AND Page zéro
26 ROL Page zéro
27 Extension future
28 PLP inhérent
29 AND #immédiat
2A ROL accumulateur
2B Extension future
2C BIT absolu
2D AND absolu
2E ROL absolu
2F Extension future
3Ø BMI relatif
31 AND (Page zéro), Y
32 Extension future
33 Extension future
34 Extension future
35 AND Page zéro, X
36 ROL Page zéro, X
37 Extension future

38	SEC inhérent
39	AND absolu, Y
3A	Extension future
3B	Extension future
3C	Extension future
3D	AND absolu, X
3E	ROL absolu, X
3F	Extension future
4Ø	RTI inhérent
41	EOR (Page zéro, X)
42	Extension future
43	Extension future
44	Extension future
45	EOR Page zéro
46	LSR Page zéro
47	Extension future
48	PHA inhérent
49	EOR #immédiat
4A	LSR accumulateur
4B	Extension future
4C	JMP absolu
4D	EOR absolu
4E	LSR absolu
4F	Extension future
5Ø	BVC relatif
51	EOR (Page zéro), Y
52	Extension future
53	Extension future
54	Extension future
55	EOR Page zéro, X
56	LSR Page zéro, X
57	Extension future
58	CLI inhérent
59	EOR absolu, Y
5A	Extension future
5B	Extension future
5C	Extension future
5D	EOR absolu, X
5E	LSR absolu, X
5F	Extension future
6Ø	RTS inhérent
61	ADC (Page zéro, X)
62	Extension future
63	Extension future
64	Extension future
65	ADC Page zéro
66	ROR Page zéro
67	Extension future
68	PLA inhérent
69	ADC #immédiat
6A	ROR accumulateur
6B	Extension future
6C	JMP (indirect)
6D	ADC absolu
6E	ROR absolu
6F	Extension future
7Ø	BVS relatif
71	ADC (Page zéro), Y
72	Extension future
73	Extension future
74	Extension future
75	ADC Page zéro, X
76	ROR Page zéro, X
77	Extension future
78	SEI inhérent
79	ADC absolu, Y
7A	Extension future
7B	Extension future
7C	Extension future
7D	ADC absolu, X
7E	ROR absolu, X
7F	Extension future

8Ø	Extension future
81	STA (Page zéro, X)
82	Extension future
83	Extension future
84	STY Page zéro
85	STA Page zéro
86	STX Page zéro
87	Extension future
88	DEY inhérent
89	Extension future
8A	TXA inhérent
8B	Extension future
8C	STY absolu
8D	STA absolu
8E	STX absolu
8F	Extension future
9Ø	BCC relatif
91	STA (Page zéro), Y
92	Extension future
93	Extension future
94	STY Page zéro, X
95	STA Page zéro, X
96	STX Page zéro, Y
97	Extension future
98	TYA inhérent
99	STA absolu, Y
9A	TXS inhérent
9B	Extension future
9C	Extension future
9D	STA absolu, X
9E	Extension future
9F	Extension future
AØ	LDY #immédiat
A1	LDA (Page zéro, X)
A2	LDX #immédiat
A3	Extension future
A4	LDY Page zéro
A5	LDA Page zéro
A6	LDX Page zéro
A7	Extension future
A8	TAY inhérent
A9	LDA #immédiat
AA	TAX inhérent
AB	Extension future
AC	LDY absolu
AD	LDA absolu
AE	LDX absolu
AF	Extension future
BØ	BCS relatif
B1	LDA (Page zéro), Y
B2	Extension future
B3	Extension future
B4	LDY Page zéro, X
B5	LDA Page zéro, X
B6	LDX Page zéro, Y
B7	Extension future
B8	CLV inhérent
B9	LDA absolu, Y
BA	TSX inhérent
BB	Extension future
BC	LDY absolu, X
BD	LDA absolu, X
BE	LDX absolu, Y
BF	Extension future
CØ	CPY #immédiat
C1	CMP (Page zéro, X)
C2	Extension future
C3	Extension future
C4	CPY Page zéro
C5	CMP Page zéro
C6	DEC Page zéro
C7	Extension future

C8	INY inhérent
C9	CMP #immédiat
CA	DEX inhérent
CB	Extension future
CC	CPY absolu
CD	CMP absolu
CE	DEC absolu
CF	Extension future
DØ	BNE relatif
D1	CMP (Page zéro), Y
D2	Extension future
D3	Extension future
D4	Extension future
D5	CMP Page zéro, X
D6	DEC Page zéro, X
D7	Extension future
D8	CLD inhérent
D9	CMP absolu, Y
DA	Extension future
DB	Extension future
DC	Extension future
DD	CMP absolu, X
DE	DEC absolu, X
DF	Extension future
EØ	CPX #immédiat
E1	SBC (Page zéro, X)
E2	Extension future
E3	Extension future
E4	CPX Page zéro
E5	SBC Page zéro
E6	INC Page zéro
E7	Extension future
E8	INX inhérent
E9	SBC #immédiat
EA	NOP inhérent
EB	Extension future
EC	CPX absolu
ED	SBC absolu
EE	INC absolu
EF	Extension future
FØ	BEQ relatif
F1	SBC (Page zéro), Y
F2	Extension future
F3	Extension future
F4	Extension future
F5	SBC Page zéro, X
F6	INC Page zéro, X
F7	Extension future
F8	SED inhérent
F9	SBC absolu, Y
FA	Extension future
FB	Extension future
FC	Extension future
FD	SBC absolu, X
FE	INC absolu, X
FF	Extension future

6 La carte de la mémoire de l'Oric

	FFFF
ROM du BASIC	
	BFE0
	BB80
Deuxième jeu de caractères	
	B800
Jeu de caractères	
	B400
RAM disponible si "GRAB" n'est pas utilisé	
	9800
Zone des programmes	
	0500
Espace système	
	0400
Adresses d'E/S	
	0300
Variables	
	0200
Pile	
	0100
Page Zéro	
	0000

Dans la même collection

Le guide du MO5 — *André Deledicq*
Initiation au Basic TO7/TO7-70 — *Christine et François-Marie Blondel*
Un ordinateur en fête — *Serge Pouts-Lajus*
Un ordinateur et des jeux — *Jean-Pascal Duclos*
Guide pratique de l'enseignement assisté par ordinateur — *Jean-Michel Lefèvre*
Faites vos jeux en assembleur sur TO7/TO7-70 — *Michel Oury*
Jeux vidéo, jeux de demain — *Georges-Marie Becherraz/Alain Graber*
Le Basic DOS du TO7/TO7-70 et du MO5 — *Christine et François-Marie Blondel*
Basic TO7, manuel de référence — *Savena*
Initiation à Logo — *Doris Avram/Michèle Weidenfeld/l'équipe de S.O.L.I.*
Logo, manuel de référence — *D. Avram/T. Savatier/M. Weidenfeld/S.O.L.I.*
Logo, des ailes pour l'esprit — *Horacio C. Reggini*
Initiation au Forth — *S.E.F.I.*
Forth, manuel de référence
Manuel de l'assembleur 6809 du TO7/TO7-70 — *Michel Weissgerber*
Manuel de l'assembleur 6809 du MO5 — *Michel Weissgerber*
La face cachée du TO7/TO7-70 — *Jean-Baptiste Touchard*
Manuel technique du TO7/TO7-70 — *Michel Oury*
Manuel technique du MO5
Macintosh, Multiplan, Macpaint — *Eddie Adamis*
Vous et l'ordinateur APPLE — *Edward H. Carlson*
Écrivons un programme pour APPLE — *Royal Van Horn*
Le Logo sur APPLE — *Harold Abelson*
Premiers pas avec le ZX-SPECTRUM — *Ian Stewart/Robin Jones*
Plus loin avec le ZX-SPECTRUM — *Ian Stewart/Robin Jones*
Le langage machine du ZX-SPECTRUM — *Ian Stewart/Robin Jones*
Guide pratique de l'ORIC-ATMOS du Basic à l'assembleur — *Bussac/Lagoutte*
Des programmes pour votre ORIC — *Michel Piot*
Premiers pas avec le COMMODORE 64 — *Ian Stewart/Robin Jones*
Musique sur COMMODORE 64 — *James Vogel/Nevin B. Scrimshaw*
Guide pratique du VIDÉOTEX et du MINITEL — *Saboureau/Bouché*
Guide pratique de l'ordinateur personnel d'IBM — *Boisgontier/Dalloz/Emery/Portefaix/Salzman*

Dépôt légal : Avril 1985
Imprimé en France par Pollina, 85400 Luçon - N° 7062

Programmer en langage machine sur Oric-Atmos et Oric-1

A travers des exemples, simples, ce livre vous permettra d'écrire vos routines en langage machine, de les stocker, de les exécuter et de les utiliser dans vos propres programmes BASIC.
Vous pourrez ainsi gagner en vitesse et en puissance en programmant en assembleur avec la même facilité qu'en BASIC.



ISBN : 2-7124-**0574**-9

F 95/85/05


PROGRAMMER EN LANGAGE MACHINE
SUR ORIC-ATMOS ET ORIC-1
cedic/nathan